UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA TÉCNICA INDUSTRIAL:
ELECTRÓNICA INDUSTRIAL

PROYECTO FIN DE CARRERA

IMPLEMENTACIÓN EN VHDL DE GENERADORES DE NÚMEROS PSEUDO-ALEATORIOS (PRNG´S) CRIPTOGRÁFICAMENTE SEGUROS

AUTOR: MANUEL GRAU SEGURA

TUTOR: ENRIQUE SAN MILLAN HEREDIA

OCTUBRE DE 2009

Índice de contenidos

Índice de figuras

Introducción del proyecto

1 Introducción del proyecto

1.1 Objetivos del proyecto

La tecnología RFID es una tecnología que está atrayendo un enorme interés y que se ha convertido en una tecnología con un despliegue y crecimiento muy amplio y elevado, debido a la enorme importancia que están cobrando los Sistemas de Autentificación Automáticos. La gran ventaja de esta tecnología radica en que transmite la información sin necesidad de contacto físico, los datos son transmitidos por radio frecuencia entre un transmisor (Transponder) y un lector.

El principal problema o tema a tratar de esta tecnología es la necesidad de disminuir al máximo posible la vulnerabilidad de la información a transmitir, de ahí surge la necesidad de utilizar unos códigos encriptados. En estas tecnologías hay una restricción importante en el tamaño del circuito que se puede utilizar, por lo que es necesario estudiar qué algoritmos resultan más adecuados para la utilización en estos sistemas.

La mayor parte de los algoritmos de encriptación necesarios están basados en la generación de una serie de números pseudoaleatorios, por lo que es fundamental estudiar diferentes algoritmos de generación de números pseudoaleatorios (PRNGs, "Pseudo-Ramdom Number Generators") y sus implementaciones adecuadas en hardware.

En este proyecto nos planteamos el estudio de varias implementaciones de varios PRNGs diferentes, analizando las prestaciones obtenidas para estos algoritmos: área ocupada por el circuito, velocidad de funcionamiento, etc.

Por tanto, establecemos una lista de objetivos de este proyecto:

1. Estudio de varios algoritmos de generación de números aleatorios.
2. Diseño hardware de los algoritmos estudiados, utilizando lenguajes de descripción de hardware (HDLs), en concreto VHDL. La implementación deberá ser lo más genérica posible, para poder realizar el correspondiente estudio para diferentes números de bits en los generadores.
3. Implementación de los diferentes algoritmos, y estudio de los principales parámetros (área, tiempo, y consumo) para diferentes números de bits. El objetivo es realizar este estudio utilizando diferentes tecnologías. Nos planteamos utilizar dos tecnologías diferentes: una de las tecnologías a utilizar estará basada en dispositivos programables (FPGAs) y la otra estará centrada en circuitos a medida (ASICs).
4. Analizar los resultados obtenidos en las diferentes tecnologías, y con los diferentes números de bits, para todos los algoritmos estudiados.

La organización del proyecto quedará establecida de la siguiente manera:

I. En primer lugar se hablará de los conceptos básicos sobre la tecnología RFID, en este apartado se realizará una introducción sobre esta tecnología y sus aplicaciones.

II. Se dará una noción de lo que es un generador de números pseudo-aleatorios, que son los algoritmos en los que se centrará el proyecto.

III. Se hablará de los conceptos básicos sobre el lenguaje utilizado en la programación, que en este proyecto será VHDL.

IV. Posteriormente se comentarán los programas empleados en el transcurso del proyecto, y a su vez se explicará la metodología de trabajo empleada para su ejecución.

V. Dada la necesidad del empleo de máquinas virtuales en el transcurso del proyecto se detallarán a conciencia y se explicará la gran

importancia de éstas no sólo para la realización del proyecto sino también de cara a un futuro.

VI. Por último se realizará una implementación exhaustiva de los diferentes algoritmos de generadores de números pseudo-aleatorios (PRNG´s) que se han considerado para este proyecto. Para cada algoritmo se implementará en vhdl, se realizará la simulación, su posterior comprobación, y a continuación se realizará la síntesis a través de las dos tecnologías mencionadas para finalmente obtener los resultados en cuanto a área, tiempo y consumo. Por último una vez obtenido los resultados de los algoritmos se realizará el análisis y la comparación de todos ellos.

Los algoritmos que se van a implementar son los siguientes:

a) El generador de números pseudo-aleatorios denominado Blum blum shub (BBS).

b) Un primer generador de números pseudo-aleatorios que llamaremos A.

c) Otro generador de números pseudo-aleatorios que denominaremos B.

d) Y una tercera opción que designaremos como C.

Estos tres últimos algoritmos son unos tipos de PRNG´s ultraligeros basados en el uso de una función triangular.

Conceptos teóricos previos

2 Conceptos teóricos previos

2.1 Tecnología RFID

La tecnología RFID (Radio Frequency IDentification) [1][2][3][4] es una tecnología de identificación automática mediante ondas de radiofrecuencia. Identificación que se consigue mediante etiquetas compuestas de una antena impresa para la transmisión/recepción de ondas electromagnéticas y un chip conteniendo un código de identificación único.

Las etiquetas RFID son unos dispositivos pequeños, similares a una pegatina, que pueden ser adheridas o incorporadas a un producto, animal o persona.

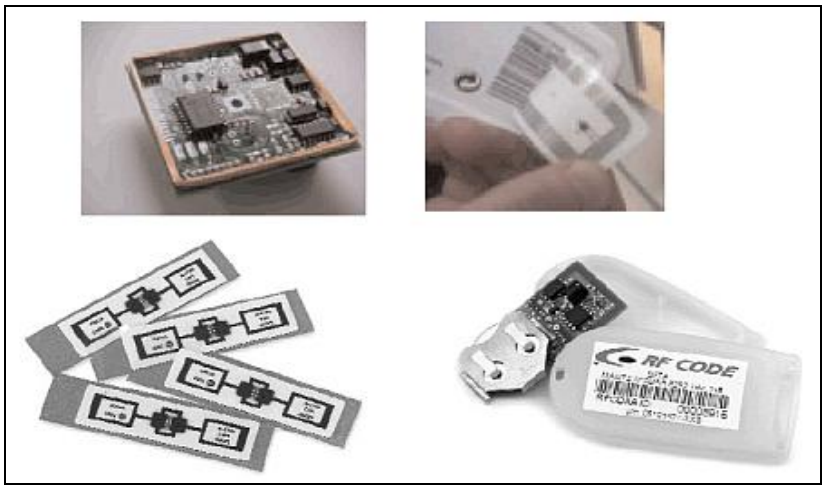

Figura 1.- Etiquetas y dispositivos RFID.

Esta tecnología no es novedosa. Desde los años 40 los militares estadounidenses utilizaron este sistema de radiofrecuencia en la Segunda

Guerra Mundial para el reconocimiento a distancia de los aviones. Sin embargo, es ahora cuando esta tecnología se ha ganado la confianza de los industriales, que ven en ella el medio de optimizar la trazabilidad de todas las mercancías en toda la cadena de distribución. Dependiendo del tamaño, tipo y antena del chip, podría rastrearse un producto desde 2 centímetros a 13 metros en los más sencillos, hasta incluso varios kilómetros en los más complejos. Son realmente pequeños y, tal y como se está avanzando en esta materia, en poco tiempo podrían ser considerados virtualmente invisibles.

Ventajas RFID

La RFID es una tecnología que comparte objetivos con la de identificación por código de barras (en la actualidad la tecnología más extendida) pero la supera en ventajas y funcionalidades:

- Con RFID, cada producto tendría un número de identificación único. Con el actual código de barras cada producto tiene el mismo número identificativo que cualquier otro del lote de producción.

- Es posible la lectura de múltiples etiquetas a la vez.

- No es necesaria visión directa, estos chips pueden ser leídos a cierta distancia y así, por ejemplo, se acabarían las colas en las cajas del supermercado porque el contenido de los carros se identificará a distancia con una sencilla lectura por radio.

- Puede tener mayor resistencia física frente a agentes externos.

- Están dotados estos chips de hasta 512 bits de memoria y una antena sensible a las ondas de radio, por lo que las etiquetas electrónicas permiten comunicar a distancia los datos que contienen sin necesidad de pila o corriente continua, ni siquiera un lector óptico, que sí es necesario para leer los códigos de barras.

Desventajas y problemas RFID

Este nuevo sistema de identificación proporciona a los comercios más información sobre los hábitos de los consumidores de la que nunca hayan podido conseguir. Como nadie ha habilitado un sistema que desactive el chip una vez cumplida su función, continuará emitiendo sus informaciones, y cualquiera dotado de un receptor será libre de localizar los productos, mucho más allá de su punto de venta, pudiendo elaborar si quisiera informes sobre los hábitos de consumo de cada familia. Sería un medio de recopilar datos sin la autorización del afectado (el consumidor en este caso), por lo que se vulnerarían los elementales principios consagrados en la Ley Orgánica de Protección de Datos de Carácter Personal.

Funcionamiento

El modo de funcionamiento de los sistemas RFID es simple. La etiqueta RFID, que contiene los datos de identificación del objeto al que se encuentra adherido, genera una señal de radiofrecuencia con dichos datos. Esta señal puede ser captada por un lector RFID, el cual se encarga de leer la información y pasarla en formato digital a la aplicación específica que utiliza RFID.

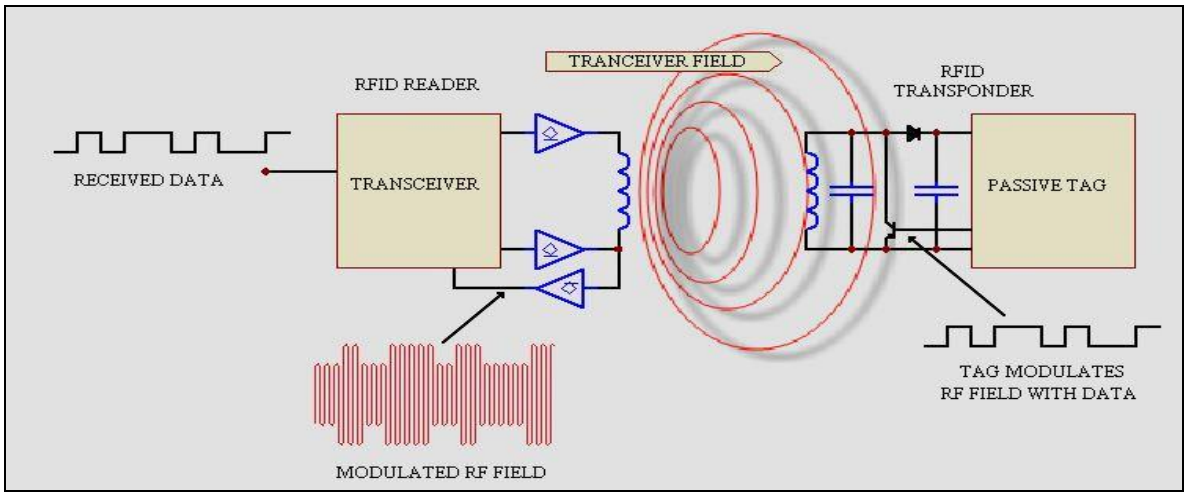

Figura 2.- Esquema de funcionamiento de un sistema RFID.

El método más común de leer las etiquetas RFID es el que se ha denominado como 'acoplamiento inductivo' en las etiquetas pasivas. En este método la antena del lector crea un campo magnético en un área cercana que llega a la etiqueta. La energía generada por este campo es utilizada por la etiqueta para devolver una señal al lector conteniendo la información almacenada en la etiqueta. El lector transmite esta información a una aplicación que se encarga de asociar el identificador almacenado en la etiqueta en cuestión con la información referente al producto al que la etiqueta se encuentra pegada. Una vez procesada, esta información se transmite a los sistemas de gestión que se encargan de actualizar la información de inventario correspondiente.

Elementos RFID

Un sistema RFID consta de los siguientes tres componentes:

- *Etiqueta RFID* o *transpondedor:* compuesta por una antena, un transductor radio y un material encapsulado o chip. El propósito de la antena es permitirle al chip, el cual contiene la información, transmitir la información de identificación de la etiqueta. El chip posee una memoria interna con una capacidad que depende del modelo y varía de una decena a millares de bytes. Esta memoria puede ser de lectura, de lectura y escritura o de anticolisión (etiquetas especiales que permiten que un lector identifique varias al mismo tiempo).

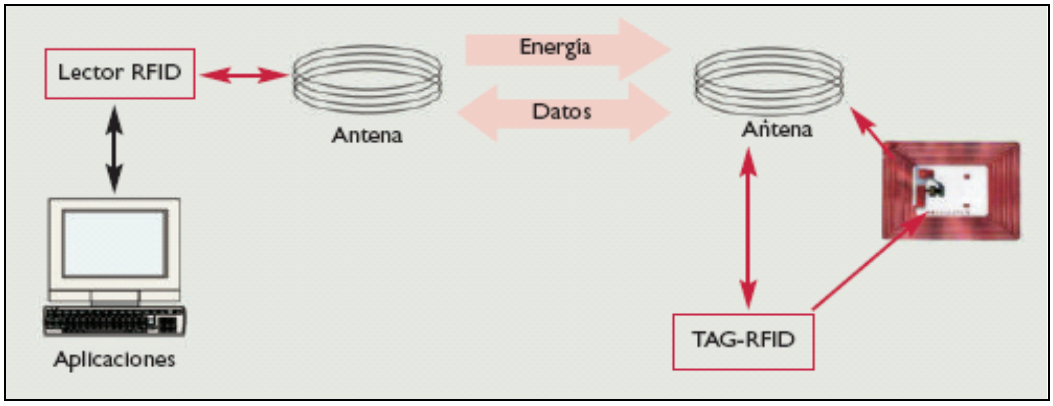

Figura 3.- Tecnología RFID, elementos.

- *Lector de RFID* o *transceptor:* compuesto por una antena, un transceptor y un decodificador. El lector envía periódicamente señales para ver si hay alguna etiqueta en sus inmediaciones. Cuando capta una señal de una etiqueta (la cual contiene la información de identificación de esta), extrae la información y se la pasa al subsistema de procesamiento de datos.
- *Subsistema de procesamiento de datos o Middleware RFID:* proporciona los medios de proceso y almacenamiento de datos.

Tipos de etiquetas RFID

Las etiquetas pasivas son las más usuales y permanecen inactivas hasta que se les solicita información por el método denominado 'acoplamiento inductivo'.

Otro tipo de etiquetas son las semi-pasivas. Estas son muy similares a las anteriores salvo que incluyen una pequeña batería que permite que el circuito integrado esté permanente alimentado y que elimina la necesidad de incorporar la antena que tome potencia de la señal inductora. Estas etiquetas responden más rápidamente y su radio de acción es mayor, por el contrario son más caras que las pasivas.

Finalmente existen etiquetas RFID activas. Su principal característica es que incorporan una fuente de energía (batería) de mayor capacidad lo que posibilita rangos de acceso mayores, memorias más grandes y poder almacenar información adicional enviada por el transmisor-receptor. En la actualidad, las etiquetas activas más pequeñas tienen el tamaño aproximado de una moneda, rangos de acceso de efectivos de hasta 10 metros y la batería puede durar hasta varios años. También son las más caras.

Frecuencias en RFID

El rango de frecuencias para esta tecnología RFID es el mostrado en la figura siguiente:

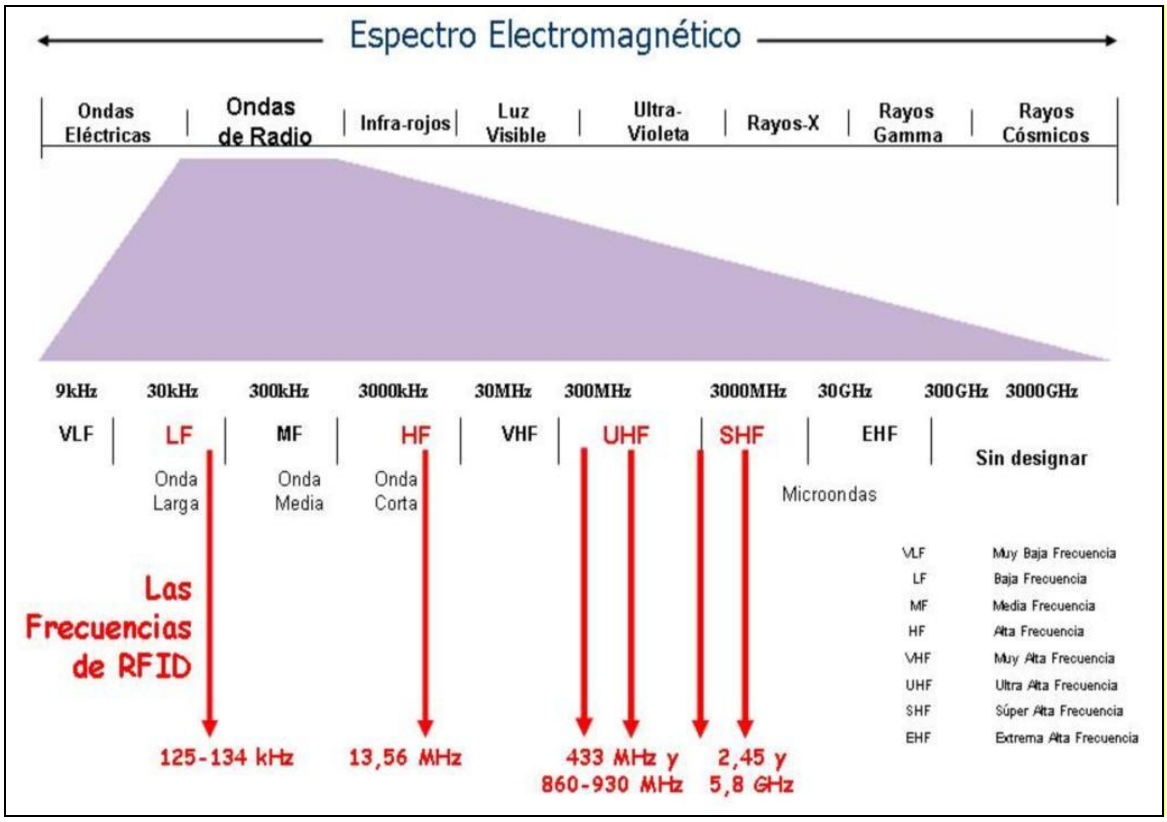

Figura 4.- Frecuencia de trabajo de las RFID.

Principales aplicaciones de la tecnología RFID

Una de las principales áreas de aplicación de la tecnología RFID es la cadena de suministro. En este campo RFID permite transformar la forma en la que se gestiona el inventario en la cadena de suministro (almacenes, transporte, estanterías y cajas) de forma automática a objetos a través de la radiofrecuencia y convirtiendo los datos que se reciben en información digital. Así se podrá ampliar la gestión la cadena de suministro mas allá de los límites corporativos, llevándola a escenarios de relaciones con clientes/proveedores en los que se mejora la visibilidad (información on-line de los procesos empresariales) y todo ellos a un coste muy asequible.

De entre los beneficios específicos del RFID en la gestión de la cadena de suministro se pueden señalar como más representativos los siguientes:

- Mayor automatización y mejora de la productividad.

- Reducción de las necesidades de almacenaje.

- Menores pérdidas en los productos almacenados.

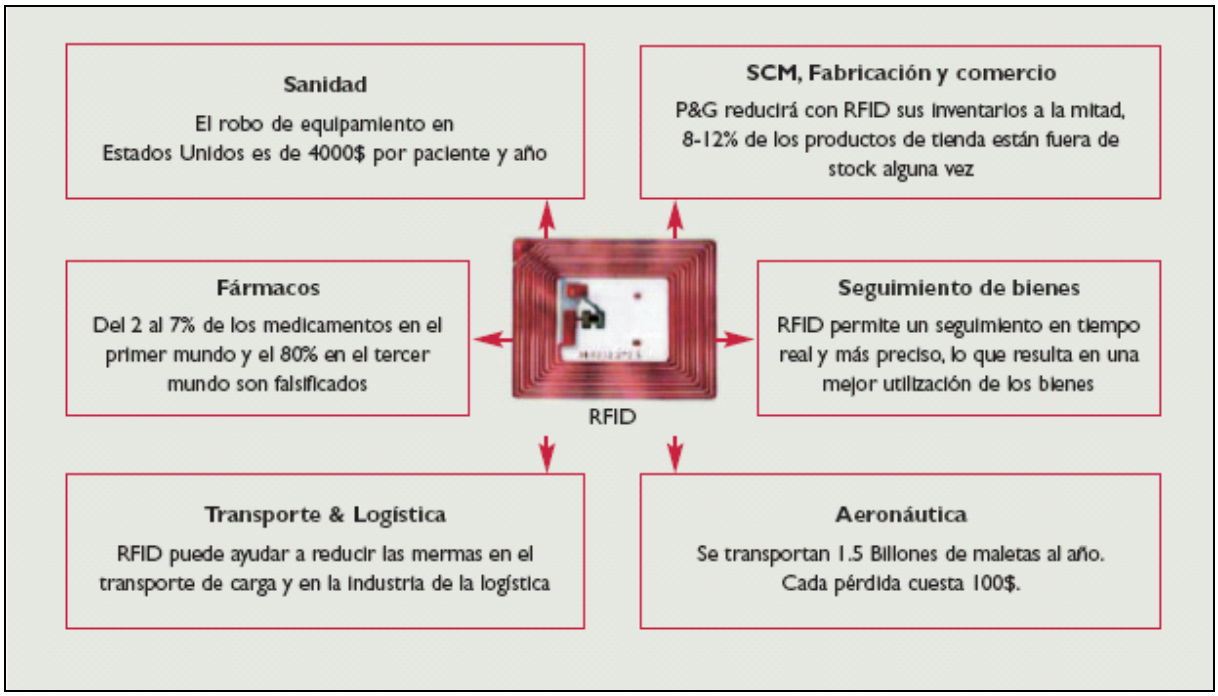

Figura 5.- Distintas aplicaciones de la tecnología RFID.

Otra de las principales aplicaciones de la RFID es el sector textil.

La aplicación de esta novedosa tecnología en los procesos operativos de empresas del sector textil, permite lecturas más rápidas y precisas de prendas y artículos textiles, además de obtener un control permanente de la mercancía y una reducción de la intervención humana, disminuyendo con ello posibilidad de error.

La tecnología RFID permite conocer en todo momento dónde se encuentra el artículo identificado (etiquetado con un tag RFID), además de información adicional como los procesos a los que ha sido sometido o la ubicación en la que se encuentra en el almacén. Con ello, se consigue una

mayor trazabilidad de los productos, así como facilidades en la realización de inventarios y la reducción de pérdidas, robos y falsificaciones.

En el caso de las tiendas textiles, disponer de información fiable de los artículos en stock que se encuentran en la tienda evitaría la pérdida de ventas debido a no conocer su ubicación en tiempo real.

Protocolo SLAP

Para poder utilizar la tecnología RFID hay que solucionar uno de los problemas que tiene esta tecnología que es el de los ataques de intrusos. Para ello se puede utilizar un protocolo de autentificación que nos sirva para proteger la información que se envía. Uno de los protocolos que podríamos utilizar sería el protocolo SLAP [14], es un protocolo de autentificación para RFID basado en modular exponentiation (ME); además para este protocolo también se necesita de un generador de números aleatorios como los que se implementan en este proyecto.

2.2 Generador de números pseudo-aleatorios

Un generador de números pseudo-aleatorios [5] es un algoritmo matemático que produce una sucesión indefinida de números aparentemente aleatorios (pseudo-aleatorios), es decir, que satisfacen, en mayor o menor grado, las pruebas habituales de aleatoriedad. Para su arranque precisan de un valor inicial denominado semilla.

La generación de números aleatorios es de trascendental importancia en criptografía; por ejemplo, en la obtención de claves criptográficas, en sistemas de autenticación pregunta-respuesta, en ciertos protocolos de autenticación fuerte, etc.

Ejemplos de estos generadores son los congruenciales y los de registro de desplazamiento de realimentación lineal (LFSR).

2.3 Lenguaje VHDL

En este proyecto para la realización de las diferentes implementaciones se ha realizado a través del lenguaje vhdl [8][9].

En lo que concierne a este apartado se hará una breve introducción de este lenguaje.

A mediados de los años setenta se produce una fuerte evolución en los procesos de fabricación de los circuitos integrados, y junto a las tecnologías bipolares, surge la MOS (metal oxide semiconductor), principalmente la NMOS, promoviendo el desarrollo de circuitos digitales hasta la primera mitad de los años ochenta. En aquellas épocas, el esfuerzo de diseño se concentraba en los niveles eléctricos para establecer características e interconexiones entre los componentes básicos a nivel de transistor. El proceso de diseño era altamente manual y tan solo se empleaban herramientas como el PSPICE para simular esquemas eléctricos con modelos previamente personalizados a las distintas tecnologías. A medida que pasaban los años, los procesos tecnológicos se hacían más y más complejos. Los problemas de integración iban en aumento y los diseños eran cada vez más difíciles de depurar y de dar mantenimiento. Inicialmente los circuitos MSI (Medium Scale Integration) y LSI (Low Scale Integration) se diseñaron mediante la realización de prototipos basados en módulos más sencillos. Cada uno de estos módulos estaba formado por puertas ya probadas, este método poco a poco, iba quedándose obsoleto. En ese momento (finales de los años setenta) se constata el enorme desfase que existe entre tecnología y diseño. La considerable complejidad de los chips que se pueden fabricar, implica unos riesgos y costes de diseño desmesurados e imposibles de asumir por las empresas. Es entonces, cuando diversos grupos de investigadores empiezan a crear y desarrollar los llamados "lenguajes de descripción de hardware" cada uno con sus peculiaridades. Empresas tales como IBM con su IDL, el TI - HDL de Texas Instruments, ZEUS de General Electric, etc., así como los primeros prototipos empleados en las universidades, empezaron a desarrollarse buscando una solución a los problemas que presentaba el diseño de los sistemas complejos. Sin embargo, estos lenguajes nunca alcanzaron el nivel de difusión y consolidación necesarias por motivos distintos. Unos, los

industriales, por ser propiedad de la empresa permanecieron encerrados en ellas y no estuvieron disponibles para su estandarización y mayor difusión, los otros, los universitarios, perecieron por no disponer de soporte ni mantenimiento adecuado. Alrededor de 1981 el Departamento de Defensa de los Estados Unidos desarrolla un proyecto llamado VHSIC (Very High Speed Integrated Circuit) su objetivo era rentabilizar las inversiones en hardware haciendo más sencillo su mantenimiento. Se pretendía con ello resolver el problema de modificar el hardware diseñado en un proyecto para utilizarlo en otro, lo que no era posible hasta entonces porque no existía una herramienta adecuada que armonizase y normalizase dicha tarea, era el momento de los HDL's.

En 1983, IBM, Intermetrics y Texas Instruments empezaron a trabajar en el desarrollo de un lenguaje de diseño que permitiera la estandarización, facilitando con ello, el mantenimiento de los diseños y la depuración de los algoritmos, para ello el IEEE propuso su estándar en 1984. Tras varias versiones llevadas a cabo con la colaboración de la industria y de las universidades, que constituyeron a posteriori etapas intermedias en el desarrollo del lenguaje, el IEEE publicó en diciembre de 1987 el estándar IEEE std 1076-1987 que constituyó el punto firme de partida de lo que después de cinco años sería ratificado como VHDL. Esta doble influencia, tanto de la empresa como de la universidad, hizo que el estándar asumido fuera un compromiso intermedio entre los lenguajes que ya habían desarrollado previamente los fabricantes, de manera que éste quedó como ensamblado y por consiguiente un tanto limitado en su facilidad de utilización haciendo dificultosa su total comprensión. Este hecho se ha visto incluso ahondado en su revisión de 1993. Pero esta deficiencia se ve altamente recompensada por la disponibilidad pública, y la seguridad que le otorga el verse revisada y sometida a mantenimiento por el IEEE. La independencia en la metodología de diseño, su capacidad descriptiva en múltiples dominios y niveles de abstracción, su versatilidad para la descripción de sistemas complejos, su posibilidad de reutilización y en definitiva la independencia de que goza con respecto de los fabricantes, han hecho que VHDL se convierta con el paso del tiempo en el lenguaje de descripción de hardware por excelencia.

2.4 Programas utilizados

Los programas que se han empleado para llevar a cabo la realización del proyecto son Modelsim [10][11] e ISE [12][13] de Xilinx, WMware Workstation y Synopsys de Cadence.

El primero de ellos se ha utilizado para la implementación del código en el lenguaje de descripción hardware vhdl y su posterior simulación.

El segundo se ha utilizado para realizar la síntesis de los diferentes circuitos implementados y de esa manera poder comparar así los diferentes algoritmos tanto en tiempo requerido como en área utilizada. Como norma general (en ciertos casos no ha sido posible y se especifica cual se ha usado) se ha elegido una FPGA SPARTAN 3 (XC3S50) para la síntesis del circuito, se ha elegido esta FPGA debido a que es una de las cuales no son grandes pero tiene el suficiente tamaño para comparar de una manera clara los diferentes circuitos que se han implementado.

El programa WMware Workstation se ha empleado para la creación de las máquinas virtuales tanto del sistema operativo Linux Suse de 64 bits como para el otro sistema operativo utilizado Solaris 10 también de 64 bits de Sun, en los que se ejecuta el Synopsys de Cadence.

Por último este programa Synopsys [16][17][18][24][25] se ha utilizado también para la síntesis de los algoritmos diseñados y poder llevar un estudio a cabo un estudio adicional de área y consumo, este estudio de área sería diferente al de ISE puesto que el ISE se utiliza para implementar en FPGA´s que son más genéricas y se miden en LUTs mientras que el Synopsys se usa para implementar en ASIC´s que son circuitos integrados hechos a la medida para un uso en particular y su diseño está basado en celdas estándares (standard cells).

2.5 Test-bench

Para la realización de la simulación de los diferentes códigos implementados se han realizado una serie de códigos estándar, como bancos de prueba, en los cuales se estimulan las señales de entrada con ciertos valores.

En este proyecto se han realizado en concreto tres tipos diferentes de test-bench:

El primero de ellos para estimular las señales de entrada del algoritmo Blum Blum & Shub; estas señales que se estimulan a través del test-bench son las señales de reloj, reset, inicio, y las entradas semilla y M, que son las que introducen los valores iniciales para el algoritmo y de las que dependerá el resultado final (según los números que introduzcamos como semilla se generarán distintos resultados).

Se ha realizado otro tipo de banco de prueba para la opción del algoritmo A, en el que las únicas señales que se estimulan en la entrada son las de inicio, reset y reloj; en este test-bench no se estimulan las valores semilla puesto que se diseñan como constantes en los tres algoritmos ultraligeros de PRNGs, se diseñan así puesto que sino habría que añadir un multiplicador como componente para realizar las primeras operaciones con estas semillas, que a su vez generarían señales de doble tamaño. Con todo esto, en un primer lugar se realizó el diseño empleando el uso del multiplicador pero dado que se obtenía una síntesis que reflejaba un elevado tamaño de área se desestimó, usando en el diseño propio del algoritmo estas constantes que no se sintetizan.

Para las opciones de los algoritmos B y C se emplea otro tipo de banco de prueba muy similar al del algoritmo A; las señales que se estimulan también en la entrada son las de inicio, reset y reloj, la variación está en que a como es debido usar los comandos *shift left* y *shift right* en estos diseños se emplea la biblioteca *numeric_std* en lugar de las bibliotecas *std_logic_unsigned* y *std_logic_arith*. Por esto, también hay que tener en cuenta que hay que definir las señales que sean vectores como *unsigned* en lugar de *std_logic_vector*.

Los códigos de estos tres tipos de bancos de prueba se muestran en el apartado 5 del anexo A, para probar las diferentes implementaciones lo único que se ha hecho es cambiar el genérico N (número de bits a la salida o la mitad del número de bits de la entrada) usando el correspondiente banco de prueba según el tipo de algoritmo en que se esté realizando la implementación.

Metodología – flujo de trabajo

3 Metodología – flujo de trabajo

3.1 Flujo de trabajo

Para llevar a cabo la ejecución de este proyecto se ha seguido una rigurosa metodología o flujo de trabajo en el que se ha ido trabajando en diferentes partes paso a paso para llevar los diferentes estudios de cada algoritmo y así poder incurrir en comparaciones óptimas entre éstos.

En primer lugar se hacía la búsqueda de estos algoritmos [32] que nos pudieran servir como generadores de números pseudo-aleatorios (puesto que aleatorios completamente del todo nunca podrían llegar a ser); una vez encontrado un generador válido se cogía un diseño de ese algoritmo para con el que tratar posteriormente.

Con ese diseño de algoritmo disponible se efectúa su implementación en lenguaje vhdl que es el que reconocerán los programas que se usarán.

Una vez comprobado que ese lenguaje vhdl es correcto y está libre de errores, se lleva a cabo la simulación mediante el programa Modelsim y se observa que no den errores en la simulación, ya que si no habría nuevamente que modificar ese código, a su vez, en la simulación se debe observar que los datos a la salida tengan coherencia con lo que deberíamos obtener.

En la siguiente figura se muestra como sería la forma óptima de implementar en vhdl con el Modelsim:

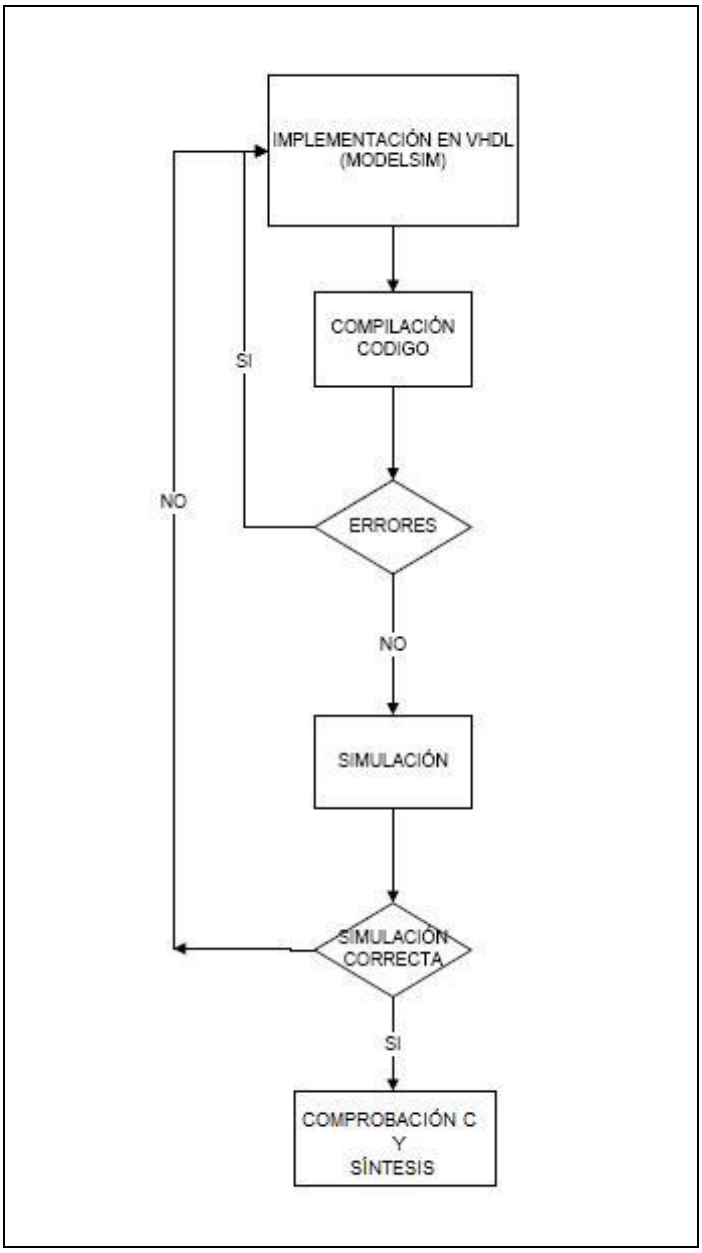

Figura 6.- Diagrama de flujo de la implementación en vhdl.

Como con los resultados obtenidos en la simulación lo único que se puede deducir es si tienen coherencia alguna con lo esperado, habría que realizar alguna comprobación de que esos datos obtenidos fuesen

completamente correctos. Para realizar esta comprobación sería inviable hacer uso de calculadoras científicas tanto por la cantidad de datos a introducir como por el elevado número de iteraciones a realizar y el consecuente riesgo de error que ello conlleva al ir introduciendo datos constantemente. Ni que decir tiene que para los casos de un número muy elevado de bits, como algunos de los que se dan en los estudios posteriores, esa comprobación podría llevar hasta meses. Por todo esto, para la comprobación se opta por la realización de un programa ejecutable mediante Turbo C++ en el que introduciendo los datos de entrada que queremos automáticamente generará esos números pseudo-aleatorios correspondiente que deberían de ser iguales a los de la simulación obtenida por el Modelsim para saber que estamos ante un código correcto de vhdl para el fin buscado de esta implementación. Este ejecutable[1] se puede probar directamente desde cualquier Pc sin necesidad de ninguna instalación previa. El único matiz de esta comprobación es que no se pueden comprobar todos los casos según los bits de entrada porque este ejecutable realizado depende de la capacidad del procesador soporte para operar con bits. Esto no daría lugar a problemas puesto que se comprueban los casos de 16 y 32 bits, y para los demás casos lo único que varía sería el número genérico N que indicaría los bits de salida.

Posteriormente se lleva a cabo la síntesis de estos códigos correctos en vhdl mediante dos procesos diferentes.

Por una parte se realiza una síntesis con el uso del programa ISE, en el que se buscará implementarlo en una FPGA y se obtendrán unos datos como el número de flip-flops, el número de slices, el área que se mide en LUTs y la frecuencia máxima que se reflejarán en el apartado de resultados, con los cuales llegaremos a una serie de conclusiones.

De otro modo se realiza otra síntesis adicional mediante el empleo del programa Synopsys en el que en lugar de implementar en FPGA´s se implementará en ASIC´s y a su vez se obtendrán otra serie de datos de área y consumo que se mostrarán también en el apartado de resultados con sus posteriores conclusiones.

[1] Se encuentra en el cd adjunto (está configurado para procesadores que usen sistemas operativos a 16 bits).

Todo este flujo de trabajo explicado se muestra mejor de una manera gráfica en la siguiente figura.

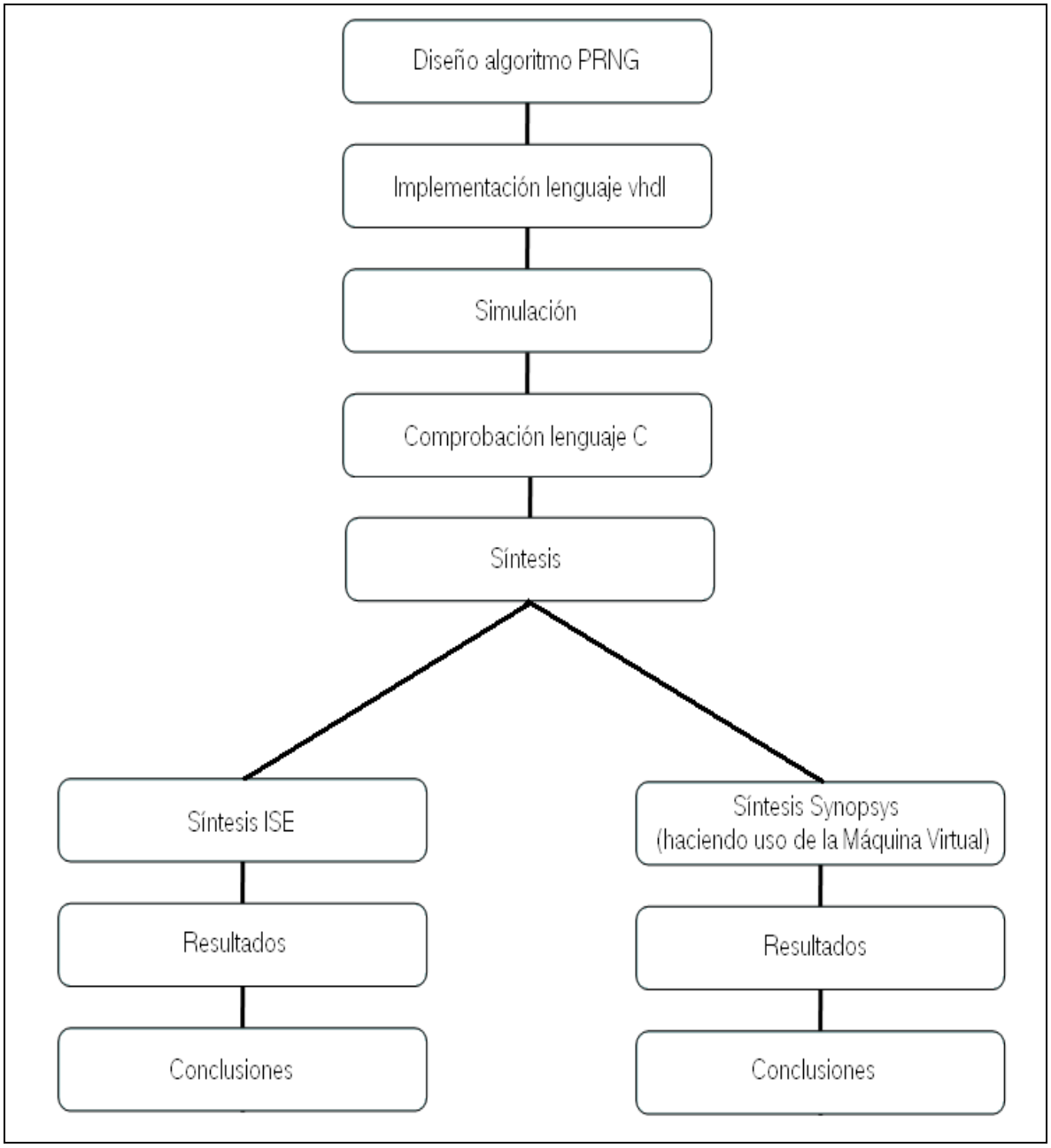

Figura 7.- Flujo de trabajo a seguir para los algoritmos.

Motivación a la utilización de SYNOPSYS

Los resultados obtenidos de las áreas del ISE y del Synopsys no se pueden comparar directamente puesto que para el caso del ISE se mide en LUTs que son unidades que se usan para implementar en una FPGA mientras que en el caso del Synopsys se usan celdas estándares (standard cells) para implementar en una ASIC.

Se ha optado por realizar esta síntesis adicional en Synopsys puesto que el programa Synopsys es mucho más profesional y usado para tal ámbito. El ISE sería más una aproximación, mientras que el Synopsys sería más fiel a la realidad a la hora de implementar puesto que las ASIC serían circuitos integrados hechos a medida para un uso en particular (de sus siglas en inglés, Circuito Integrado para Aplicaciones Específicas).

Las FPGA (**F**ield **P**rogrammable **G**ate **A**rrays, matriz de puertas programables) contienen bloques de lógica programable e interconexiones programables que permiten a un modelo de FPGA ser usada en muchas aplicaciones distintas y son reprogramables por el propio usuario. Por lo que para los diseños más pequeños o con volúmenes de producción más bajos, las FPGAs pueden tener un costo menor que un diseño equivalente basado en ASIC, debido a que el costo fijo (el costo para preparar una línea de producción para que fabrique un ASIC en particular) es muy alto, especialmente en las tecnologías más densas, más de un millón de dólares para una tecnología de 90nm (la usada en este proyecto) o menor.

Con esto tenemos que las FPGA son útiles y más baratas para diseños sencillos, y además tienen la ventaja de ser programables por el propio usuario en lugar de fabricadas a medida para cada aplicación. Pero también tenemos que si nos adentramos en un diseño más especifico o en un diseño totalmente a la medida (full custom circuits) es fundamental el uso de las ASIC que para diseños puramente digitales, las librerías de "celdas estándares" pueden ofrecer ventajas considerables en términos de costos y desempeño junto a un bajo riesgo, además las herramientas de layout automático son rápidas y fáciles de usar, y ofrecen la posibilidad de optimizar manualmente cualquier aspecto que limite el desempeño del diseño, de ahí la necesidad de que el estudio mediante el programa Synopsys sea bastante conveniente.

Para el empleo del programa Synopsys es necesaria la creación de máquinas virtuales en las que se use otros sistemas operativos diferentes al Windows puesto su uso en él no era posible con las versiones disponibles.

En los siguientes apartados se comentarán como se crearon estas máquinas virtuales.

3.2 Preparación Máquinas virtuales

Dada la gran importancia del empleo de las máquinas virtuales en este proyecto para la utilización del programa Synopsys se comenta como se crean estas máquinas virtuales paso a paso mediante el programa VMware Workstation.

Comentar que el uso de estas máquinas virtuales es muy beneficioso no sólo de cara a la inmediata utilización en este proyecto, en donde es sencillo arrancar desde el punto en que se hubiese quedado la sesión anterior y que si hubiese cualquier problema surgido bastaría con cargar una copia que se tuviese de la máquina virtual. Pero en donde verdaderamente útil el uso de estas máquinas virtuales es para el futuro de cara a utilizaciones para posteriores proyectos o trabajos en los que sea necesario el uso del programa Synopsys.

El programa Synopsys para ponerlo en funcionamiento necesita de un laborioso trabajo previo y mediante el uso de las máquinas virtuales es directamente utilizable, ahorrando una gran cantidad de tiempo. Esto es debido a su gran ventaja, y es que son completamente portables, y se podría utilizar para muy diversos fines ya sea en la universidad o en casas particulares; bastaría con tener una copia de la máquina virtual y cargarla en el programa VMware Workstation.

3.2.1 Creación de máquina virtual con sistema operativo Linux Suse para 64 bits

A continuación en este apartado se va a explicar paso a paso la creación de una máquina virtual con el sistema operativo Linux Suse para un procesador 64 bits con la ayuda de las pantallas correspondientes.

En primer lugar se arrancaría el programa WMware Workstation y se abría la opción de New Virtual Machine.

Destacar que se posee de una copia portable de una máquina virtual con este sistema operativo con el programa Synopsys ya puesto en funcionamiento para las futuras ocasiones en que sea requerido el uso de este programa tan útil en la creación de diseños digitales para la implementación en ASIC´s.

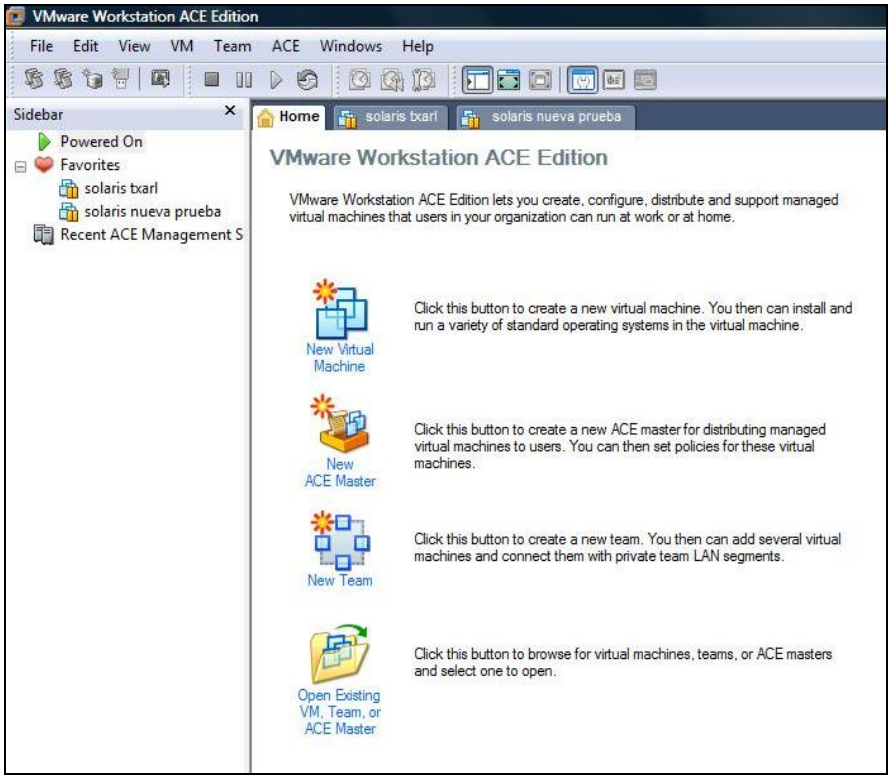

Figura 8.- Creación Máquina Virtual, paso 1.

Se abriría un asistente de la creación de una nueva máquina virtual y deberíamos pulsar en siguiente.

Figura 9.- Creación Máquina Virtual, paso 2.

Seleccionaríamos la opción Typical para nuestra máquina virtual.

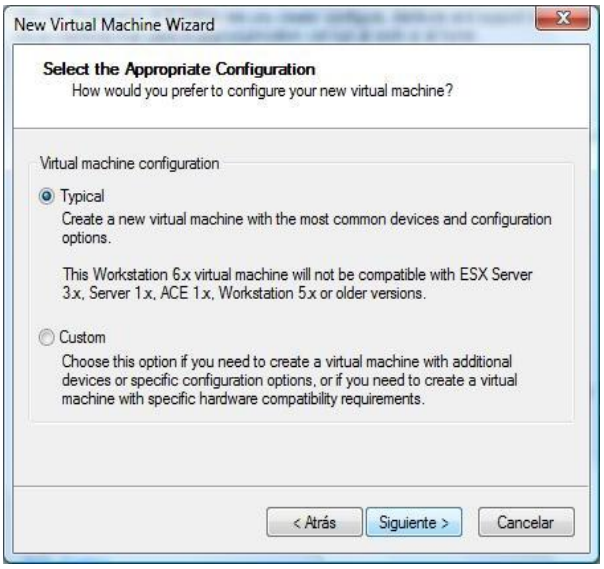

Figura 10.- Creación Máquina Virtual, paso 3.

Se marcaría la opción de Linux dentro del listado de sistemas operativos disponibles y se seleccionaría la versión correspondiente a la SUSE Linux de 64-bits.

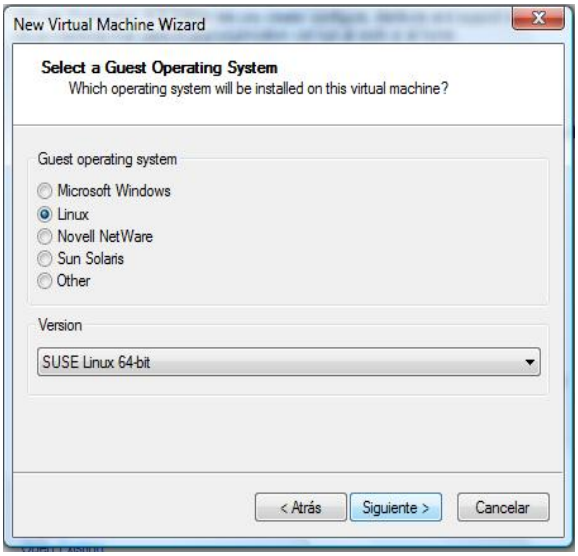

Figura 11.- Creación Máquina Virtual, paso 4.

Se daría el nombre correspondiente a la máquina virtual creada y se pondría la ruta de localización.

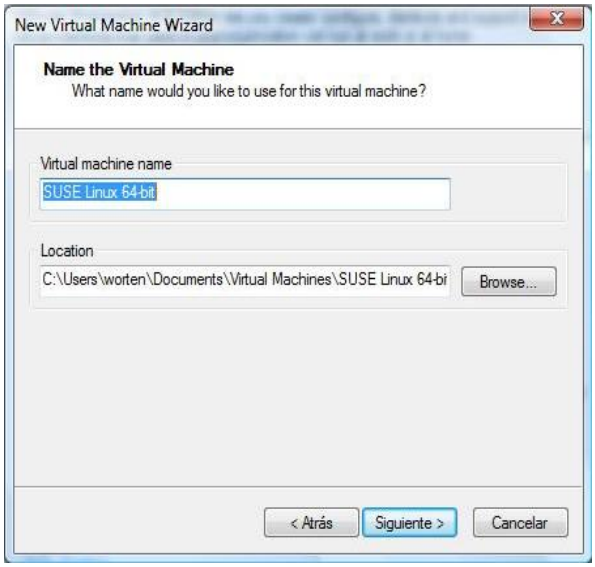

Figura 12.- Creación Máquina Virtual, paso 5.

Se seleccionaría el modo de acceso de conexión a la red, en este caso escogemos el *bridged* aunque la opción *NAT* también podría ser válida.

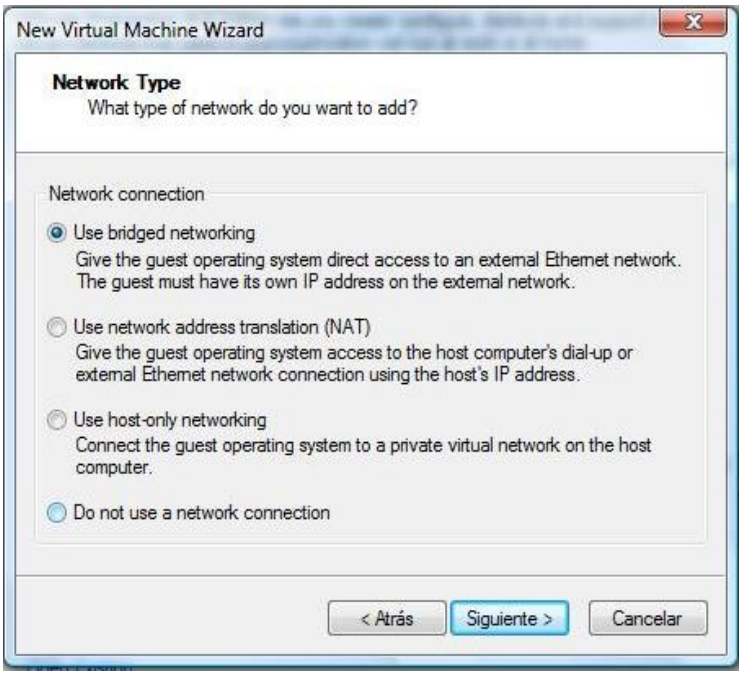

Figura 13.- Creación Máquina Virtual, paso 6.

Lo siguiente que se debería hacer es escoger la capacidad del disco duro de la máquina virtual a crear.

Es importante seleccionar bien el tamaño del disco, puesto que si fuera necesario más memoria de disco duro para almacenar archivos en la máquina virtual ya no se podría ampliar y no podríamos seguir operando si necesitásemos más memoria, para el caso de la máquina virtual creada para ser portable en ocasiones futuras se ha seleccionado una capacidad de 24 Gb, no siendo demasiados puesto que una creada y con el funcionamiento del programa Synopsys sólo quedarían libres unos 7 Gb en memoria de disco.

Marcando la opción *Allocate all disk space now* guarda y reserva automáticamente en ese momento toda la capacidad de disco del PC usada en la máquina virtual, sino se marca esta opción la reserva de memoria del disco duro del Pc irá creciendo a medida que crezca el tamaño de la máquina virtual.

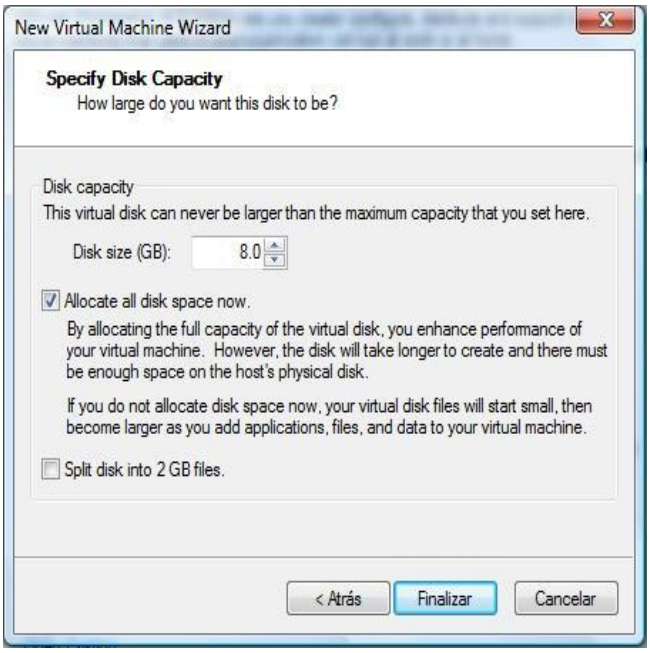

Figura 14.- Creación Máquina Virtual, paso 7.

Con todo esto ya tendríamos creada satisfactoriamente la máquina virtual necesaria.

Figura 15.- Creación Máquina Virtual, paso 8.

Aunque por último, faltaría establecer las opciones que queramos adaptarle para la máquina virtual, ya sea de memoria RAM utilizada, dispositivos, número de procesadores, etc.

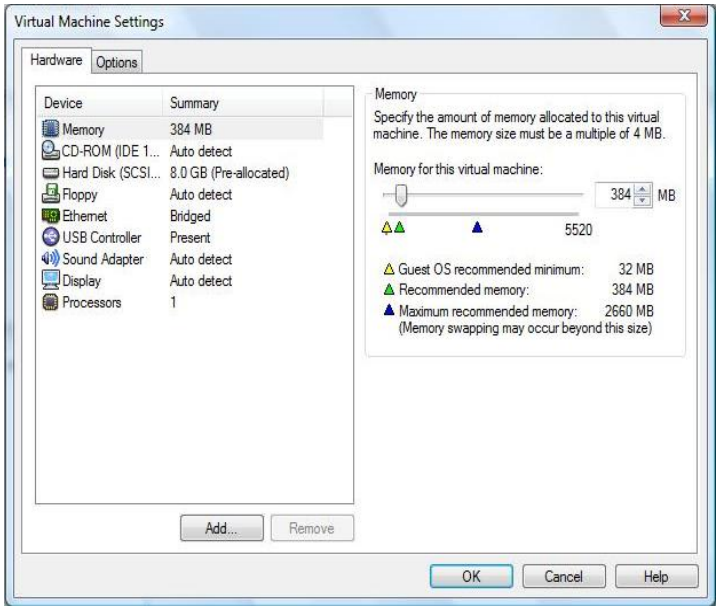

Figura 16.- Creación Máquina Virtual, paso 9.

Aunque la opción verdaderamente importante es la de indicarle a la máquina virtual de donde debe cargar el sistema operativo utilizado. Para ello dentro de las *Virtual Machine Settings* en la pestaña de *Hardware* seleccionaremos el Cd-Rom y seleccionaremos el modo de conexión a la carga del propio sistema operativo. Para la máquina portable creada usaremos la conexión mediante una imagen ISO, qué es más rápida que mediante cd, y ubicaremos la ruta en donde se encuentre esa imagen ISO dentro del Pc propio.

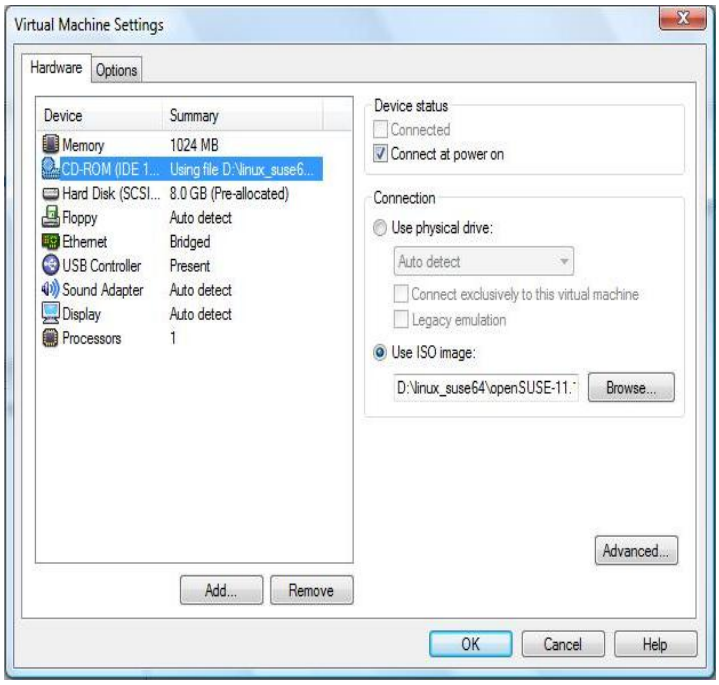

Figura 17.- Creación Máquina Virtual, paso 10.

Así ya se observa como se tendría la máquina virtual lista para su uso.

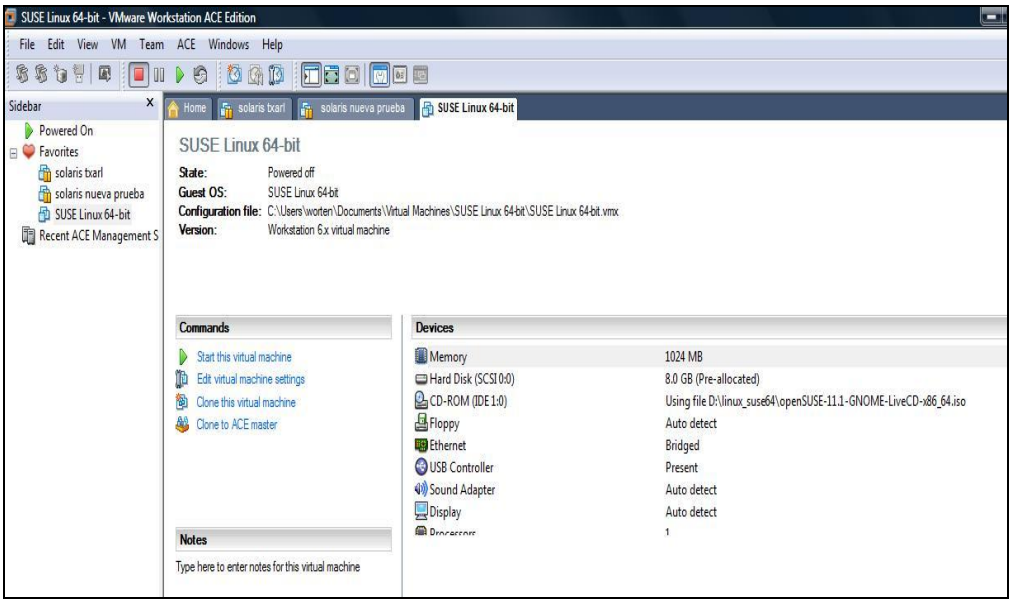

Figura 18.- Máquina Virtual creada.

Cabe destacar que es muy necesaria la necesidad de tener conexión a red en las máquinas virtuales creadas puesto que para el funcionamiento del programa Synopsys es necesario que se conecte a la red para comprobar la licencia de uso del programa.

Para el caso de esta máquina virtual con Linux Suse resulta sencilla la conexión a red puesto que se dispone de un asistente detecta automáticamente las redes para su conexión.

3.2.2 Creación de máquina virtual con sistema operativo Sun Solaris 10

En este apartado se va explicar brevemente con sus particularidades como se realiza la creación de una máquina virtual con un sistema operativo Sun Solaris 10 [29], el cual también podría ser óptimo para el uso de diversos paquetes del programa Synopsys.

Para la creación de esta máquina virtual sería una creación similar a la anterior con la salvedad de que se debería escoger la opción apropiada para este sistema operativo con su respectiva versión

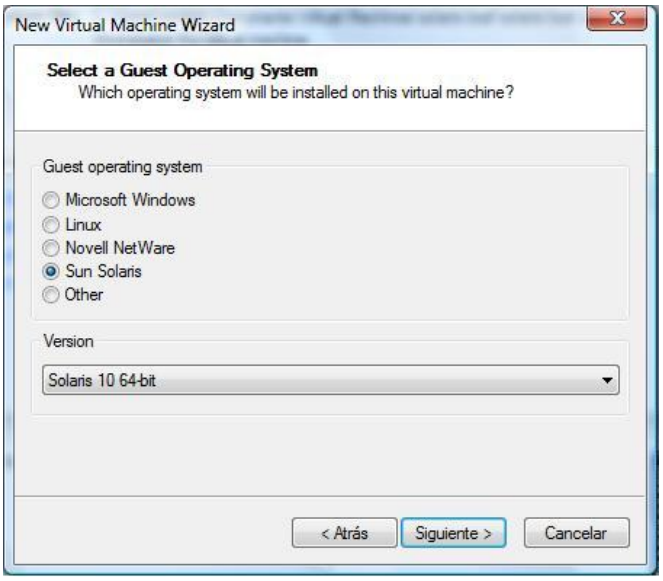

Figura 19.- Creación Máquina Virtual, modificación S.O.

También sería importante añadir la imagen ISO desde donde se cargue este sistema operativo y ubicar la ruta correspondiente de donde se encuentre en el propio Pc en las *Virtal Machine Settings* al igual que en el proceso del apartado anterior.

Sin embargo, lo verdaderamente distinto en este proceso de la creación de la máquina virtual en Solaris es que la configuración para el acceso a la red a través de Solaris [28][29][30] es verdaderamente complicada y nada intuitiva como en el caso de Linux suse.

De modo resumido se muestra en la siguiente figura la serie de comandos que se deberían introducir en una shell de Solaris para conseguir esa configuración de acceso a la red necesaria para la utilización de los diversos paquetes del programa Synopsys debida a la siempre requerida licencia de acceso.

```
# dladm show-dev
pcn0            vínculo: unknown        velocidad: 0     Mbps   dúplex: unknown
# ifconfig pcn0
pcn0: flags=1000842<BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
        inet 127.0.0.1 netmask ff000000 broadcast 127.255.255.255
        ether 0:c:29:6c:7d:1
# ifconfig pcn0 down
# ifconfig pcn0
pcn0: flags=1000842<BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
        inet 127.0.0.1 netmask ff000000 broadcast 127.255.255.255
        ether 0:c:29:6c:7d:1
# ifconfig pcn0 plumb
ifconfig: SIOCSLIFNAME for ip: pcn0: already exists
# ifconfig pcn0 192.168.1.40 netmask 255.255.255.0
# ifconfig pcn0
pcn0: flags=1000842<BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
        inet 192.168.1.40 netmask ffffff00 broadcast 192.168.1.255
        ether 0:c:29:6c:7d:1
# ifconfig pcn0 up
# ifconfig pcn0
pcn0: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
        inet 192.168.1.40 netmask ffffff00 broadcast 192.168.1.255
        ether 0:c:29:6c:7d:1
# hostname solaris
# route add default 192.168.1.1
add net default: gateway 192.168.1.1
# ping solaris
```

Figura 20.- Comandos en el terminal para la configuración de red en Sun Solaris 10.

Destacar que el comando dlad show-dev mostraría la placa de red que tenemos y la que debemos configurar.

En solaris se trata a las placas de red como "cañerías". Si corremos ifconfig vamos a ver si está conectada, para conectarla usamos el comando plumb, después asignamos el router y finalmente comprobamos si hay conexión que debería dar *is alive*.

Además de estos comandos es muy importante que previamente en el archivo */etc/resolv.conf* se le añada el número del nameserver (ip del servidor dns) y del router, a su vez, en el archivo */etc/nsswitch.conf* al lado de host file hay que escribir *dns*.

Para editar esos archivos y escribir los correspondientes datos es necesario el uso de un editor de textos, por ejemplo, el editor vi[2].

3.3 Puesta en funcionamiento Synopsys

La puesta en funcionamiento del programa Synopsys utilizado en este proyecto resulta complicada, por ello se ha dedicado este apartado para su explicación detallada.

La ventaja de la creación de las máquinas virtuales previas radica en que este apartado no sería necesario realizarlo cada vez que se quiera utilizar el programa Synopsys, sino que con la copia portable de la máquina virtual se podría ya ejecutar directamente desde cualquier Pc con la única necesidad previa de la instalación del programa VMware Workstation.

En primer lugar se debería de disponer de los respectivos paquetes del programa Synopsys a usar, ya bien porque se disponga de alguna versión o porque se haya obtenido a través del fabricante Synopsys a través de su página web.

Para la puesta en funcionamiento llevada a cabo en este proyecto se disponía del paquete FEV (Synopsys Front End & Verification tools suites) en la versión syn_vB-2008.09.

Dentro de este directorio se debe buscar el Synopsys apropiado para el sistema operativo en el que hayamos creado la máquina virtual, en este caso sería para el Linux SUSE de 64 bits. Ahora estos archivos los deberemos pasar a la máquina virtual mediante el uso de por ejemplo un dispositivo USB.

Se procederá a la extracción en la máquina virtual de dichos archivos que vienen comprimidos en formato .gzip y empaquetados en .tar, esto se hace

[2] Para conocer mejor el uso de este editor de archivos se ha incluido un Anexo C con unos comandos básicos.

ubicándose dentro de un terminal en el directorio en el que se quiera extraer los archivos mediante los siguientes comandos:

```
% ungzip -xvf toolversion(syn_vB-2008.09)_suse64.tar.gzip
% tar -xvf toolversion(syn_vB-2008.09)_suse64.tar
```

A su vez también se procederá a la extracción en el mismo directorio de unos archivos que son comunes y son necesarios para cualquier sistema operativo en el que se utilice, esto se haría de la misma manera introduciendo los comandos en el terminal:

```
% ungzip -xvf toolversion(syn_vB-2008.09)_common.tar.gzip
% tar -xvf toolversion(syn_vB-2008.09)_common.tar
```

Es necesario obtener la licencia de utilización de producto y tener conexión a la red para que el programa Synopsys a través de esa conexión se dirija a la licencia cada vez que sea necesario en su uso. Para este proyecto se disponía de una licencia de la universidad, también se puede obtener licencia a través de Synopsys [27] en su página web.

Dentro del paquete obtenido o descargado habrá una carpeta con el nombre de *installer* con unos archivos los cuales descomprimiremos en el mismo directorio de los archivos anteriores de la misma manera usando los comandos:

```
% ungzip -xvf installer_v2.0.tar.gzip
% tar -xvf installer_v2.0..tar
```

El script de instalación de synopsys utiliza la Shell del sistema "CShell" (csh), por lo que es necesario que esté instalada en el sistema. La distribución SUSE no incluye esta Shell por defecto, por lo que hay que instalarla. Existe una versión mejorada de esta Shell, "TCShell" (tcsh), que mantiene la compatibilidad con la CShell estándar.

Por comodidad se puede configurar esta nueva Shell como interfaz por defecto para el usuario de instalación, y utilizar el archivo ".cshrc" (detallado más adelante) en el directorio de la cuenta del usuario para definir variables del sistema, como la ruta de instalación de Synopsys (path) o las variables correspondientes a la licencia del programa (LM_LICENSE_FILE). De este modo, al iniciar la Shell el sistema cargará estas variables automáticamente.

A continuación se insertará en la nueva shell modificada el siguiente comando (dentro de la ruta en donde se encuentren los archivos extraídos) con el que conseguiremos que nos aparezca una interfaz gráfica con la que proseguir con la instalación [15][22][26]:

```
% ./installer.sh  -gui
```

Una vez entremos en la interfaz gráfica deberemos seguir las pantallas paso a paso de tal manera que marquemos en la instalación las opciones de los archivos para el sistema operativo que se usa y la de los archivos comunes indicando en el proceso la ruta en la que se encuentran estos archivos.

Es muy importante que para esta instalación nos encontremos como usuario en el root puesto que es necesario para algunos permisos internos del sistema operativo y para la creación de algunas carpetas. Durante la instalación guiada por el interfaz gráfico también es necesario establecer una ruta para donde se instale los productos y archivos de Synopsys en la que dentro del root hubiese una carpeta con el nombre de usr y dentro de esta otra carpeta con el nombre de synopsys (la ruta donde instalar sería /usr/synopsys).

Una vez finalizada esta instalación con la interfaz gráfica, es necesario crear un archivo con un editor de textos de Unix (por ejemplo el vi[3]), este archivo se debería encontrar en la misma ruta que los archivos que se extrajeron previamente. Este nuevo archivo creado es mencionado anteriormente con el nombre de .cshrc (el punto indica que es un archivo oculto) y en él se deberán introducir las variables de entorno con las rutas correspondientes y dejar indicada la licencia con la que podrá ejecutarse el Synopsys, sería un texto similar al siguiente:

```
setenv LM_LICENSE_FILE ___@___.uc3m.es (licencia)
setenv SNPSLMD_LICENSE_FILE ___@___.uc3m.es
setenv synopsys_root /usr/synopsys/B-2008.09
setenv SYNOPSYS /usr/synopsys/B-2008.09
set path=(/usr/synopsys/B-2008.09/bin $path)
```

Una vez escritas estas variables de entorno, rutas y establecida la licencia se guardarían, y a continuación en el terminal se ejecutaría el siguiente comando para hacer efectivos estos cambios:

```
% source .cshrc
```

Para observar que todo se ha procesado correctamente deberíamos ver lo escrito anteriormente al introducir el siguiente comando:

```
% cat .cshrc
```

[3] Ver Anexo C para los comandos en la creación del archivo

Para observar que se tiene acceso a la licencia usada mediante la conexión de red establecida se podría usar el comando ping para ver si se recibe correctamente:

```
% ping -a  ___.uc3m.es  (licencia)
```

Por último, es muy importante introducir los archivos .synopsys_dc.setup y .synopsys_vss.setup pegándolos en la siguiente ruta donde se indicó para la instalación (en la interfaz gráfica):

```
/usr/synopsys/B-2008.09/admin/setup
```

Estos archivos se podrían encontrar en el paquete del que se dispondría o se descargase o bien, también se podría editar con la información correspondiente.

Ahora ya situándose en la ruta del escritorio (Desktop) dentro de un terminal introduciendo el siguiente comando se estaría ya ejecutando el programa:

```
% design_vision
```

Este Design Vision sería la herramienta del Synopsys que se usará para este proyecto. A veces también puede ser necesario el ajuste del display en el terminal.

La herramienta estaría ya ejecutándose pero todavía no funcionaría correctamente, faltaría establecer adecuadamente las bibliotecas que se usarán en el diseño.

Para esto deberíamos editar nuevamente el archivo .synopsys_dc.setup en donde se le indique correctamente las bibliotecas a usar mediante las rutas

donde se encuentren y los archivos que se quieran usar de esas bibliotecas (archivos .db) y los símbolos (.sdb). Para este proyecto se ha usado una biblioteca con tecnología de 90nm, con lo que en la modificación de la parte correspondiente a la edición de este archivo debería quedar de la siguiente manera:

```
set search_path [list . ${synopsys_root}/libraries/syn/nano] (ruta)
set target_library saed90nm_min.db
set link_library saed90nm_min.db
set symbol_library generic.sdb
```

Ahora ya se guardarían estas modificaciones y con todo esto la herramienta Design Vision del Synopsys estaría funcionando correctamente y únicamente habría que centrarse en el funcionamiento del propio Design Vision [19][20][21][23].

Finalmente destacar que como para este proyecto se utilizan diseños con un elevado número de bits podría surgir con estos un problema, para solucionar este problema bastaría con aumentar el valor de la variable *hdlin_while_loop_iterations*, esto se hace dentro de la opción setup en las *tcl variables.*

DISEÑO PRNG´S

(PSEUDO-RANDOM NUMBER GENERATION)

DISEÑO PRNG´S

(PSEUDO-RANDOM NUMBER GENERATION)

Blum Blum Shub

ALGORITMO – IMPLEMENTACIÓN VHDL – SIMULACIÓN – SÍNTESIS ISE Y SYNOPSYS – RESULTADOS – CONCLUSIONES

4 Diseño PRNG´s (Pseudo-Random Number Generation)

4.1 Blum blum shub

4.1.1 Algoritmo

El algoritmo que usaremos a continuación para generar los números aleatorios será el Blum Blum and Shub [6][7]. Este algoritmo fue propuesto por Lenore Blum, Manuel Blum y Michael Shub en 1986. El algoritmo es el siguiente:

$$X_{n+1} = X_n^2 \mod M$$

Donde $M=pq$ es el producto de dos números primos muy grandes p y q. En cada paso del algoritmo, se obtiene un resultado para X_n, el resultado es por lo general o bien el bit de paridad de X_n ó uno ó más de los bits menos significativos de X_n. Para calcular el X_0 se utilizará un valor denominado semilla el cual será el X_{-1} de la sucesión, para posteriormente calcular todos los sucesivos valores de la sucesión.

Los dos números primos, p y q, deben ser ambos congruentes a 3 (mod 4) (esto asegura que cada residuo cuadrático posee una raíz cuadrada que también es un residuo cuadrático) y mcd ($\varphi(p-1)$, $\varphi(q-1)$) debe ser pequeña (esto hace que la longitud del ciclo sea extensa).

Para implementar este algoritmo se ha usado la implementación del algoritmo de multiplicación modular Montgomery.

Se ha considerado un nivel de seguridad con un módulo de 128 bits, a continuación se muestra la implementación para el generador de números primos usando el algoritmo Blum Blum and Shub.

4.1.2 Implementación en código Vhdl

A continuación se muestra la entidad implementada en vhdl:

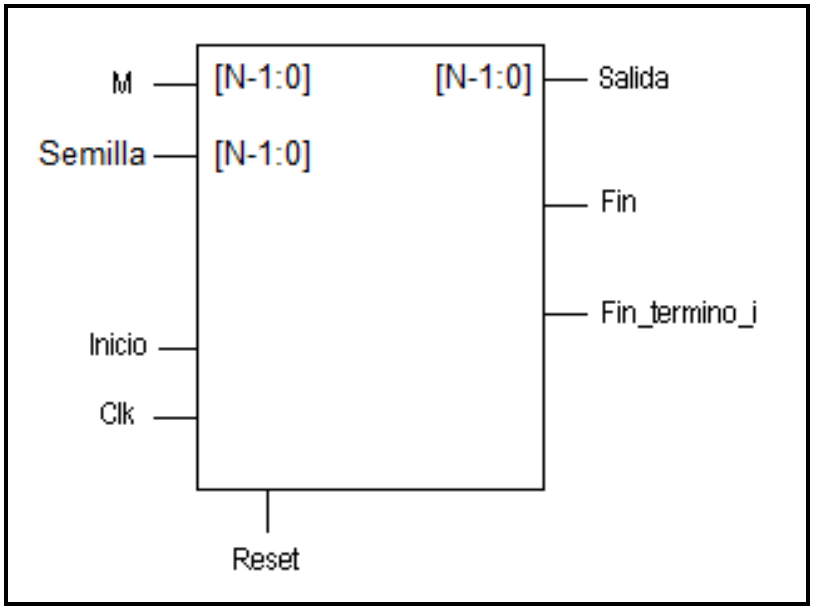

Figura 21.- Entidad Blum Blum Shub.

El bit Fin_termino_i muestra el fin de cada término de la sucesión una vez que se ha obtenido el número, en cambio el bit Fin indica el final de la sucesión. Activando el bit Inicio empieza a generar números aleatorios.

El algoritmo implementado se rige por el funcionamiento de la siguiente máquina de estados:

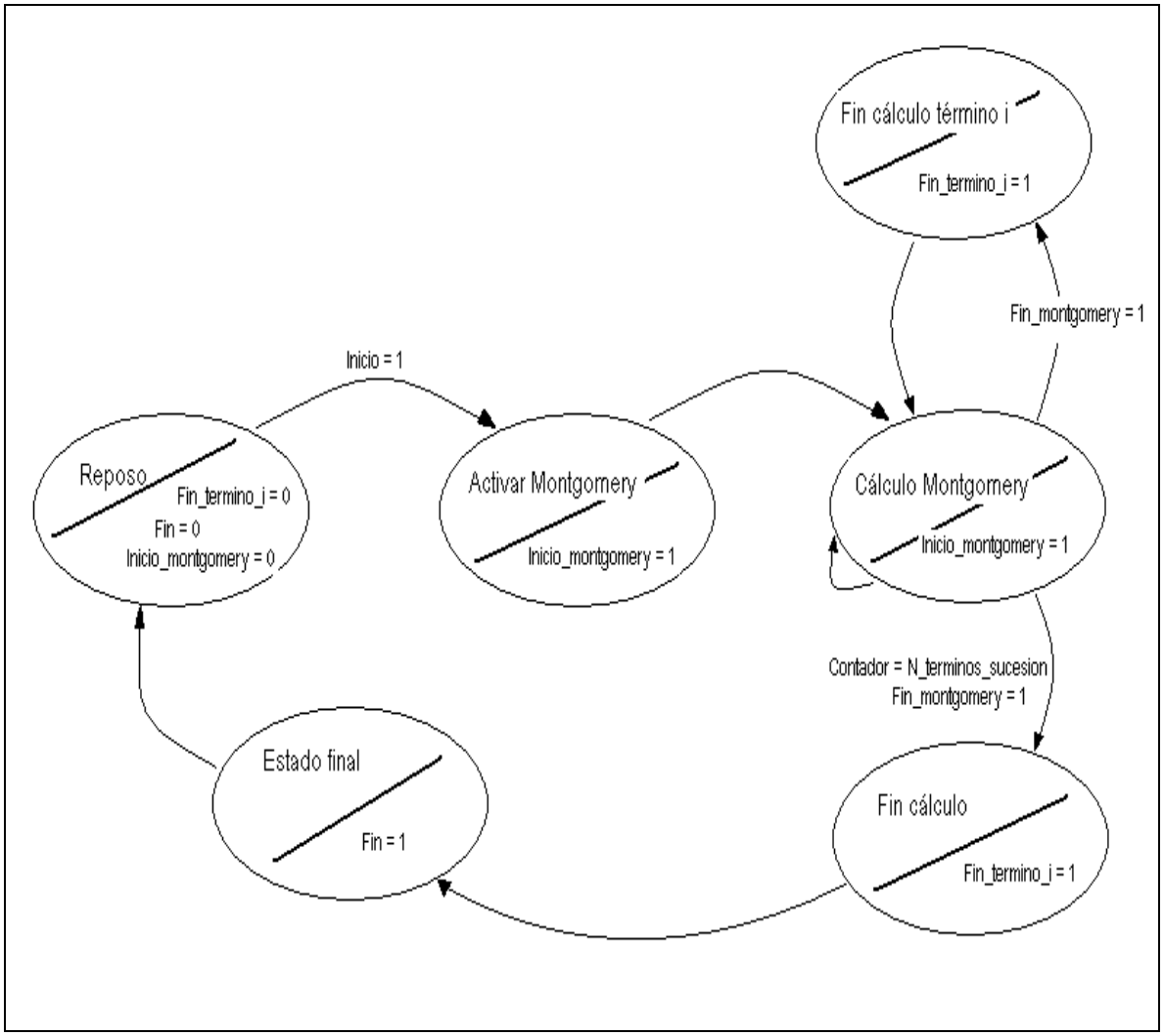

Figura 22.- Diagrama de estados Blum Blum Shub.

a) REPOSO: Está a la espera del bit de Inicio para comenzar el proceso.

Si es 1 go to Iterando, sino mantiene estado

b) ACTIVAR MONTGOMERY: Se activa el multiplicador Montgomery.

Go to Cálculo Montgomery

c) CÁLCULO MONTGOMERY: Se van realizando las multiplicaciones.

   Incrementa contador

   Si contador = N_términos_sucesión y fin_montgomery = 1 go to Fin cálculo, sino si fin_montgomery = 1 go to Fin cálculo término_i

d) FIN CÁLCULO TÉRMINO i: Estado para ir calculando los sucesivos términos de la sucesión.

   Activa la salida fin_término_i

   Go to Cálculo Montgomery

e) FIN CÁLCULO: Estado que da por concluido la actuación del multiplicador Montgomery

   Activa la salida fin_término_i

   Go to Estado final

f) ESTADO FINAL: Finaliza el proceso.

   Activa la salida fin

   Go to Reposo

En el apartado 1 del anexo A se puede observar el código vhdl que se ha utilizado para implementar este algoritmo. Para las diferentes implementaciones según el nível de seguridad empleado bastaría con cambiar el valor del genérico N según el número de bits.

El número total de términos (N_terminos_sucesion) que queremos calcular, será el número total de números aleatorios que queremos calcular, este número de términos está condicionado por el valor del contador que será una constante la cual podemos modificar.

En el código se observa que en el estado Cálculo montgomery se realizará montgomery y pasará al estado Fin cálculo término i, estos dos

estados se realizarán tantas veces como números aleatorios queramos calcular. Cuando llegamos al último término de la sucesión se pasará directamente al estado Fin cálculo donde terminará de generar números aleatorios.

El diagrama de flujo se muestra a continuación:

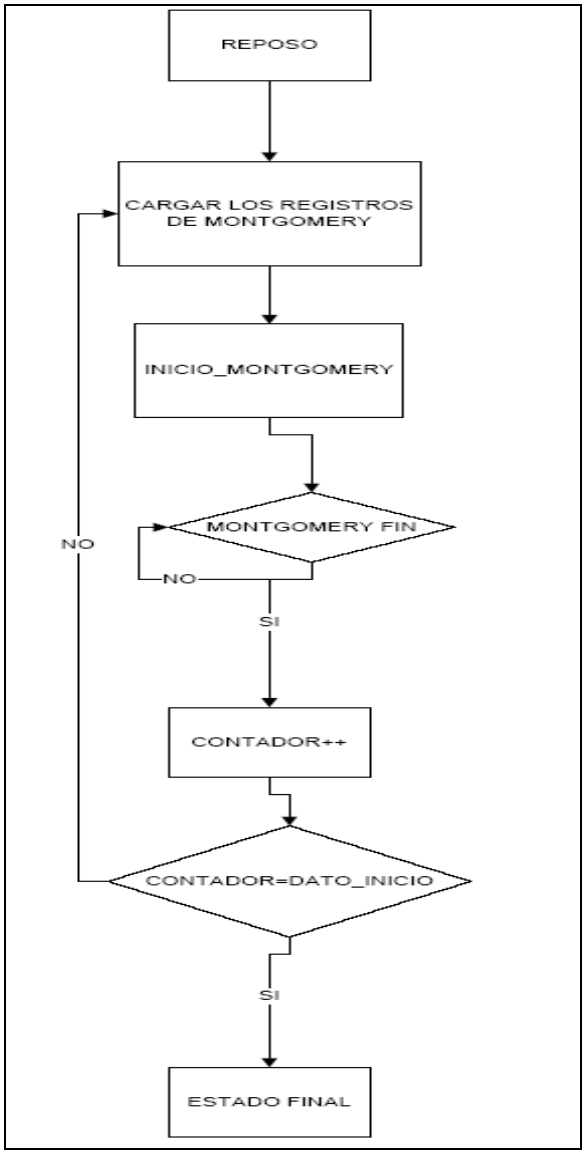

Figura 23.- Diagrama de flujo Blum Blum Shub.

4.1.3 Simulación

En la siguiente simulación se mostrará detalladamente los pasos que se realizará en el circuito. Hemos tomado como números primos n = 233 y p = 373 => por lo tanto tenemos como módulo => M = 86909 , y como termino raíz será X_{-1}= 555.

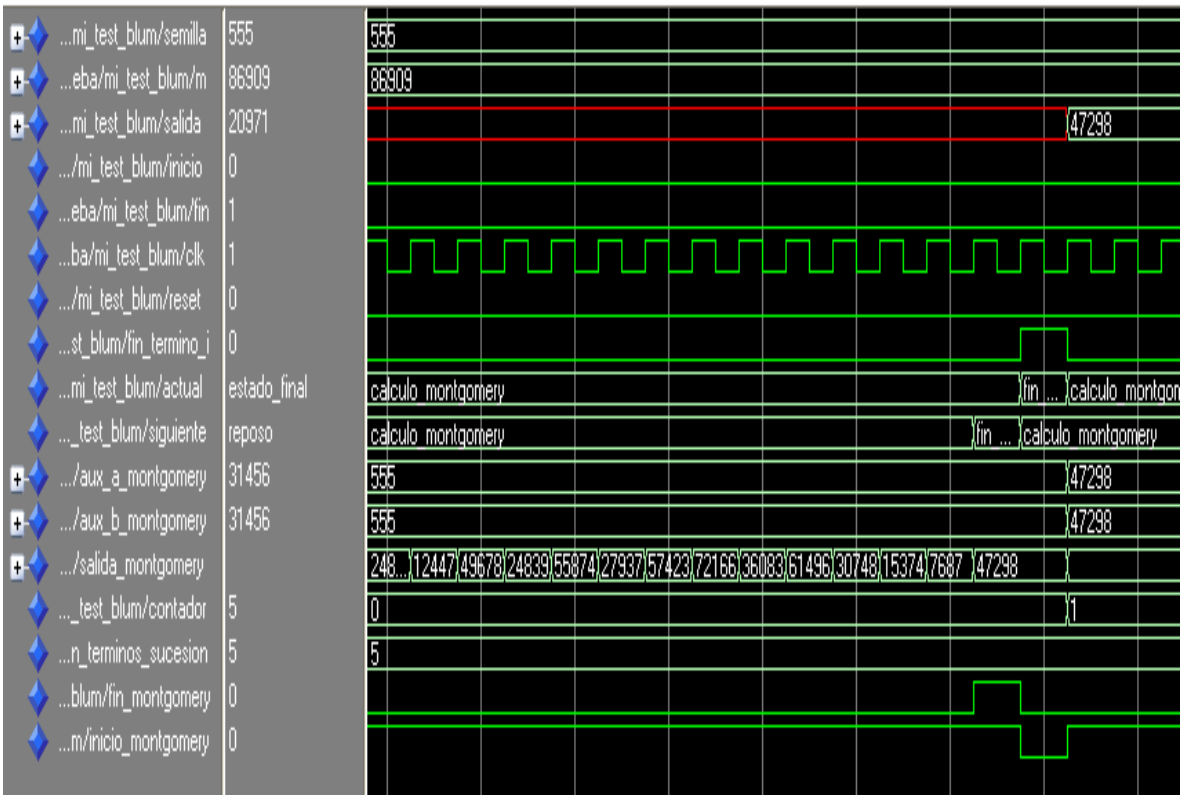

Figura 24.- Simulación 1 Bum Blum Shub.

Se puede observar como cada vez que se quiere realizar Montgomery se activa el bit inicio_montgomery, y el circuito de Montgomery cada vez que obtiene el resultado activa el bit fin_montgomery. El contador muestra el término de la sucesión que se está calculando.

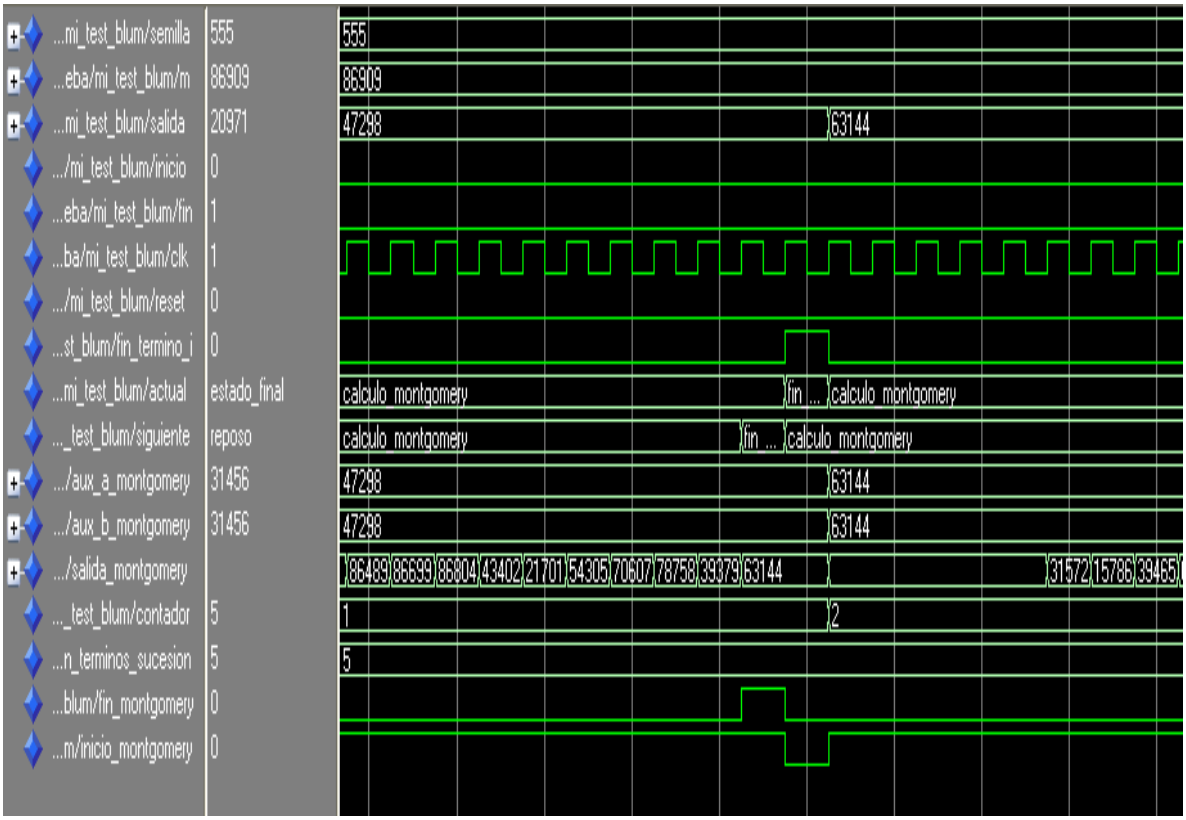

Figura 25.- Simulación 2 Blum Blum Shub.

Se irá incrementando el contador hasta que este sea igual al número de los términos de la sucesión (n_terminos_sucesion) que indicará que ya ha calculado los X números de la sucesión, y por tanto pasará al estado Fin Cálculo.

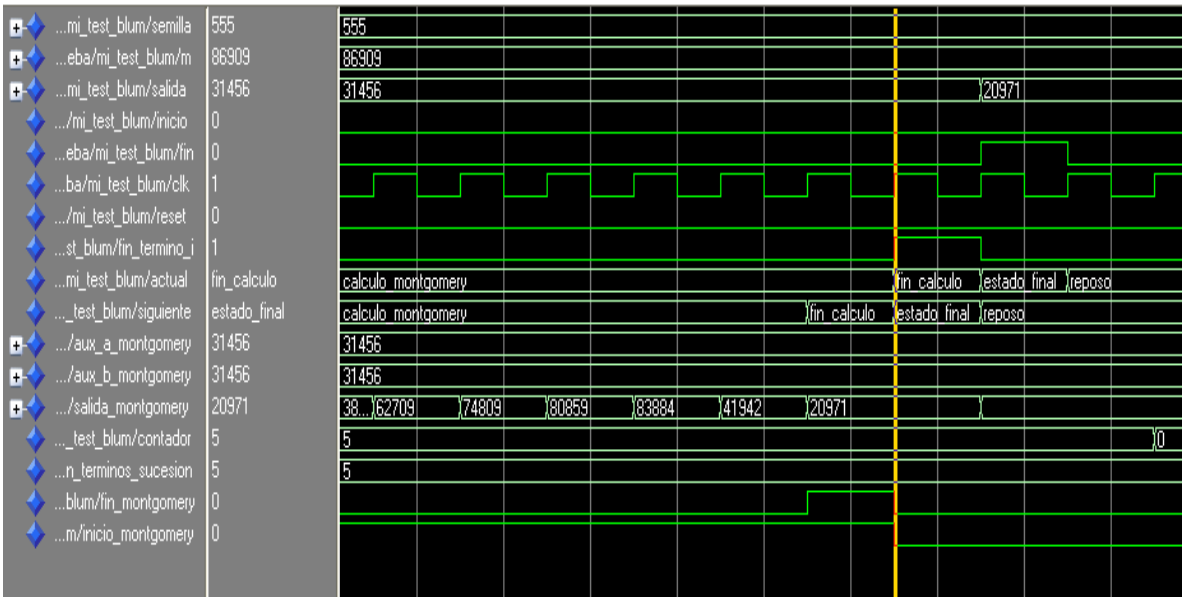

Figura 26.- Simulación 3 Blum Blum Shub.

Se muestra la simulación al completo:

Simulación para 5 términos, n = 233 y p = 373 => M = 86909, X_{-1} = 555.

Se puede observar que se llegaría al final de la simulación (fin = 1) en un tiempo de 15.6 ms.

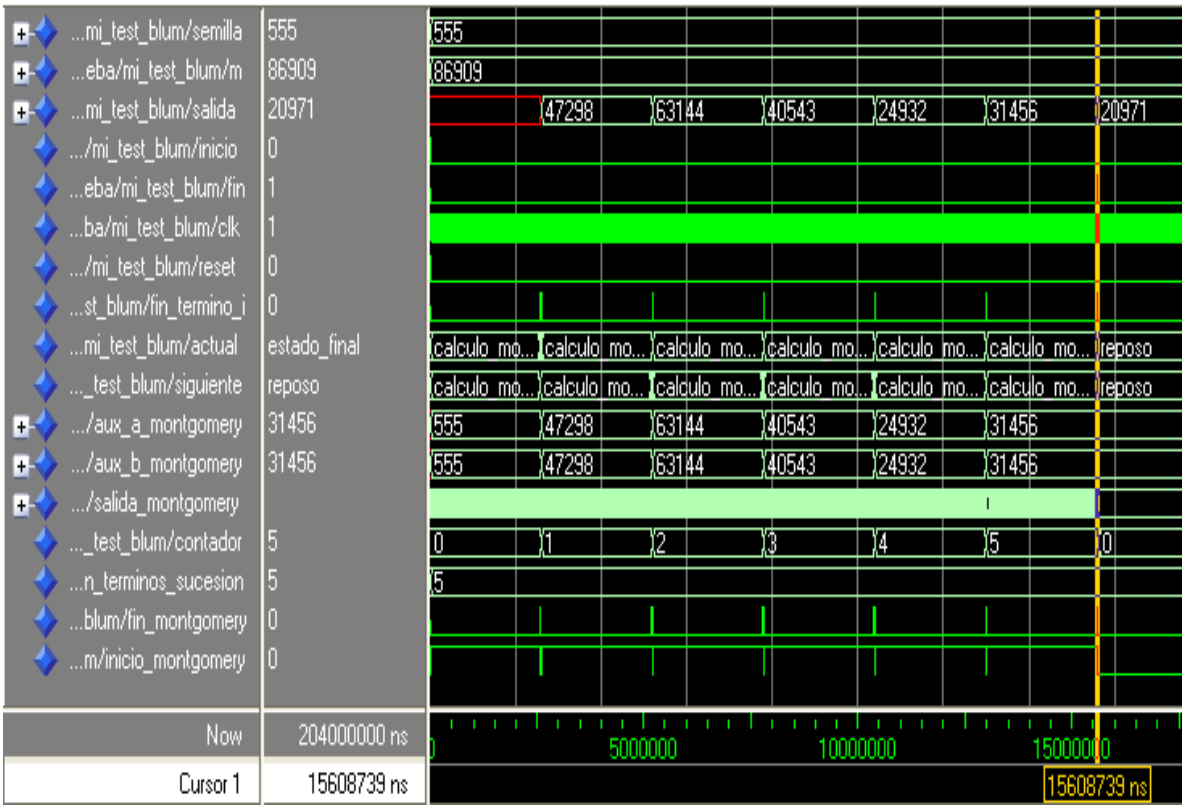

Figura 27.- Simulación 4 Blum Blum Shub.

Comprobamos estos resultados obtenidos con la siguiente tabla para ver si son correctos. En la última columna se representa la paridad de cada término en el caso de que una vez terminado con el algoritmo de Blum Blum and Shub se quiere obtener un resultado a partir de la paridad de estos términos.

X(i)	n	p	M = n x p	producto X(i)* X(i)	Núm. Generado	Paridad	
0	233	373	86909	308025	47298	0	
1	233	373	86909	2237100804	63144	0	
2	233	373	86909	3987164736	40543	1	
3	233	373	86909	1643734849	24932	0	
4	233	373	86909	621604624	31456	0	
5	233	373	86909	989479936	20971	1	
						100100	36

4.1.4 Síntesis

4.1.4.1 Síntesis ISE

A continuación se muestra la síntesis realizada de este diseño por medio del programa ISE.

Se ha buscado implementar este diseño en una FPGA lo más optima posible, que para este caso sería lo más pequeña posible.

La síntesis se realiza además de para este nivel de seguridad de 128 bits usado en el diseño previo, para los casos de 16, 256 y 512 bits para poder comparar así con los resultados obtenidos de los demás algoritmos, buscando una optimización tanto en área como en tiempo.

N = 16 bits de nível de seguridad

Para este diseño de nível de seguridad se ha escogido una FPGA de la familia Spartan 3 cuyo dispositivo es el XC3S50, que será el más empleado a lo largo de este proyecto debido a su reducido tamaño.

Property Name	Value
Product Category	General Purpose
Family	Spartan3
Device	XC3S50
Package	PQ208
Speed	-5

Implementado con optimización en Área

Frecuencia máxima = 82.649 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	127	1,536	8%
Number of 4 input LUTs	308	1,536	20%
Logic Distribution			
Number of occupied Slices	172	768	22%
Number of Slices containing only related logic	172	172	100%
Number of Slices containing unrelated logic	0	172	0%
Total Number of 4 input LUTs	323	1,536	21%
Number used as logic	308		
Number used as a route-thru	15		
Number of bonded IOBs	53	124	42%
IOB Flip Flops	16		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 98.070 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	127	1,536	8%
Number of 4 input LUTs	311	1,536	20%
Logic Distribution			
Number of occupied Slices	174	768	22%
Number of Slices containing only related logic	174	174	100%
Number of Slices containing unrelated logic	0	174	0%
Total Number of 4 input LUTs	326	1,536	21%
Number used as logic	311		
Number used as a route-thru	15		
Number of bonded IOBs	53	124	42%
Number of GCLKs	1	8	12%

N = 128 bits de nível de seguridad

Para este diseño de nível de seguridad se ha escogido una FPGA de la familia Spartan 3 cuyo dispositivo es el XC3S4000, se trata de un dispositivo más grande que el XC3S50, el que más emplearemos más a menudo dentro de lo posible en nuestro estudio, ya que en este no cabía.

Property Name	Value
Product Category	General Purpose
Family	Spartan3
Device	XC3S4000
Package	FG676
Speed	-5

Implementado con optimización en Área

Frecuencia máxima = 36.212 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	918	55,296	1%
Number of 4 input LUTs	2,281	55,296	4%
Logic Distribution			
Number of occupied Slices	1,275	27,648	4%
Number of Slices containing only related logic	1,275	1,275	100%
Number of Slices containing unrelated logic	0	1,275	0%
Total Number of 4 input LUTs	2,408	55,296	4%
Number used as logic	2,281		
Number used as a route-thru	127		
Number of bonded IOBs	389	489	79%
IOB Flip Flops	128		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 41.802 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	926	55,296	1%
Number of 4 input LUTs	2,288	55,296	4%
Logic Distribution			
Number of occupied Slices	1,354	27,648	4%
Number of Slices containing only related logic	1,354	1,354	100%
Number of Slices containing unrelated logic	0	1,354	0%
Total Number of 4 input LUTs	2,415	55,296	4%
Number used as logic	2,288		
Number used as a route-thru	127		
Number of bonded IOBs	389	489	79%
Number of GCLKs	1	8	12%

N = 256 bits de nível de seguridad

Para este diseño de nível de seguridad se ha escogido una FPGA de la familia Spartan 3 cuyo dispositivo es el XC3S5000.

Property Name	Value
Product Category	General Purpose
Family	Spartan3
Device	XC3S5000
Package	FG1156
Speed	-5

Implementado con optimización en Área

Frecuencia máxima = 23.344 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,822	66,560	2%
Number of 4 input LUTs	4,529	66,560	6%
Logic Distribution			
Number of occupied Slices	2,541	33,280	7%
Number of Slices containing only related logic	2,541	2,541	100%
Number of Slices containing unrelated logic	0	2,541	0%
Total Number of 4 input LUTs	4,791	66,560	7%
Number used as logic	4,529		
Number used as a route-thru	262		
Number of bonded IOBs	773	784	98%
IOB Flip Flops	256		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 24.636 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,832	66,560	2%
Number of 4 input LUTs	4,880	66,560	7%
Logic Distribution			
Number of occupied Slices	3,007	33,280	9%
Number of Slices containing only related logic	3,007	3,007	100%
Number of Slices containing unrelated logic	0	3,007	0%
Total Number of 4 input LUTs	5,142	66,560	7%
Number used as logic	4,880		
Number used as a route-thru	262		
Number of bonded IOBs	773	784	98%
Number of GCLKs	1	8	12%

N = 512 bits de nível de seguridad

Para este diseño de nível de seguridad se ha escogido una FPGA al igual que para la implementación anterior de la familia Spartan 3 siendo el dispositivo el XC3S5000.

Implementado con optimización en Área

Frecuencia máxima = 11.344 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	3,626	66,560	5%
Number of 4 input LUTs	9,024	66,560	13%
Logic Distribution			
Number of occupied Slices	5,061	33,280	15%
Number of Slices containing only related logic	5,061	5,061	100%
Number of Slices containing unrelated logic	0	5,061	0%
Total Number of 4 input LUTs	9,543	66,560	14%
Number used as logic	9,024		
Number used as a route-thru	519		
Number of bonded IOBs	1,541	784	196%
IOB Flip Flops	512		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 12.433 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	3,648	66,560	5%
Number of 4 input LUTs	9,199	66,560	13%
Logic Distribution			
Number of occupied Slices	5,506	33,280	16%
Number of Slices containing only related logic	5,506	5,506	100%
Number of Slices containing unrelated logic	0	5,506	0%
Total Number of 4 input LUTs	9,718	66,560	14%
Number used as logic	9,199		
Number used as a route-thru	519		
Number of bonded IOBs	1,541	784	196%
Number of GCLKs	1	8	12%

4.1.4.2 Resultados ISE

De la síntesis realizada con el programa ISE se han obtenido los siguientes resultados resumidos en las siguientes tablas:

En primer lugar los obtenidos con un esfuerzo alto en área:

	16 bits	128 bits	256 bits	512 bits
Nº flip-flops	127	918	1822	3626
Nº slices	172	1275	2541	5061
Nº LUTs	323	2408	4791	9543
Freq. Máx. (MHz)	82,649	36,212	23,344	11,344

A continuación los obtenidos con un esfuerzo alto en tiempo:

	16 bits	128 bits	256 bits	512 bits
Nº flip-flops	127	926	1832	3648
Nº slices	174	1354	3007	5506
Nº LUTs	326	2415	5142	9718
Freq. Máx. (MHz)	98,07	41,802	24,636	12,433

La siguiente gráfica mostraría una comparación entre los dos tipos de optimizaciones en función de los parámetros estudiados para el caso de 128 bits:

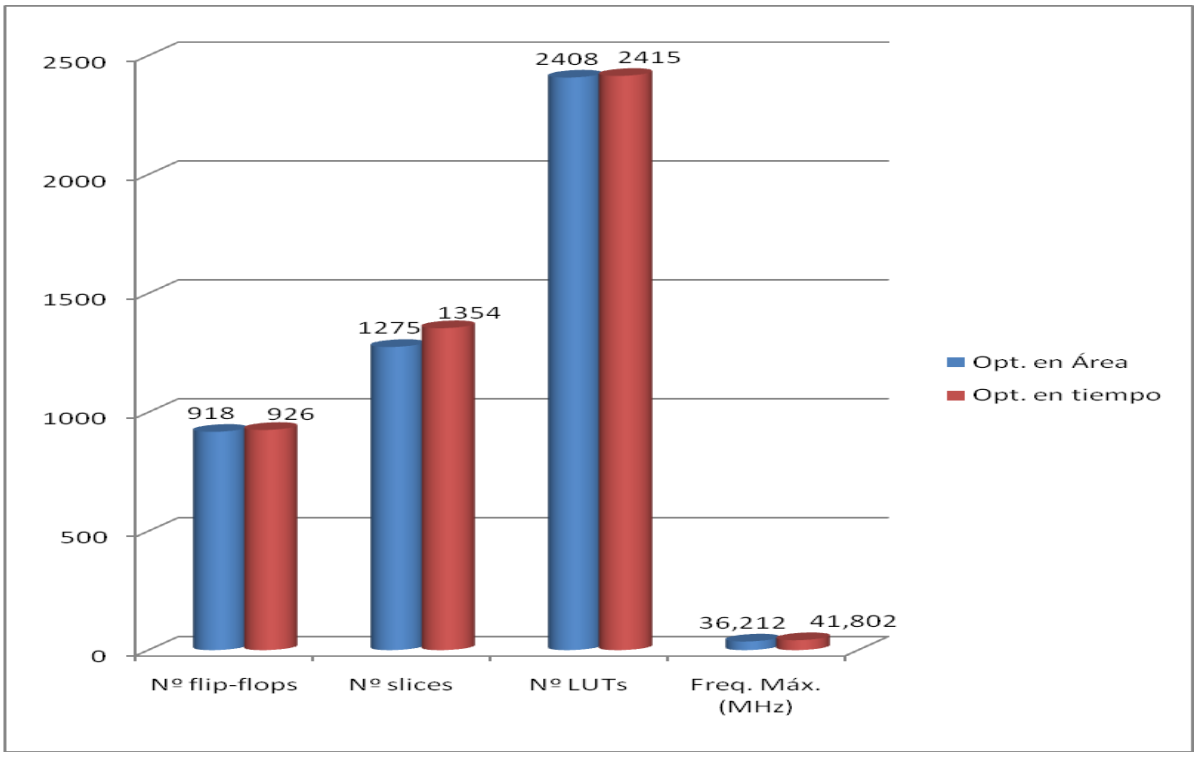

Figura 28.- Gráfica comparativa entre las optimizaciones en Área y Tiempo.

En las siguientes gráficas se mostrarían el conjunto de los resultados obtenidos para la implementación con el esfuerzo alto en área.

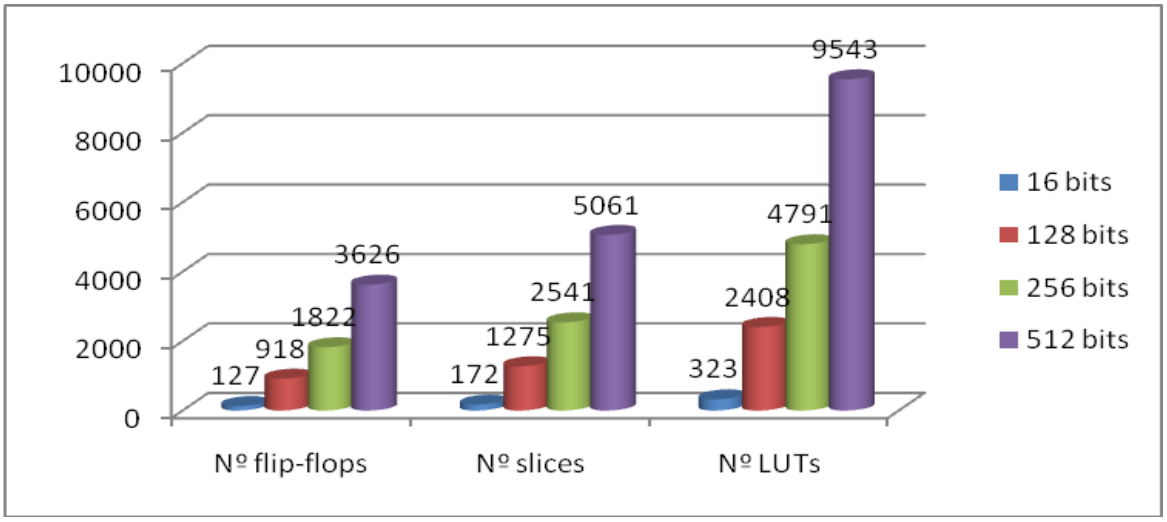

Figura 29.- Gráfica del Blum Blum Shub con el nº de flip-flops, slices y tamaño en LUTs en función del nivel de seguridad con optimización en Área.

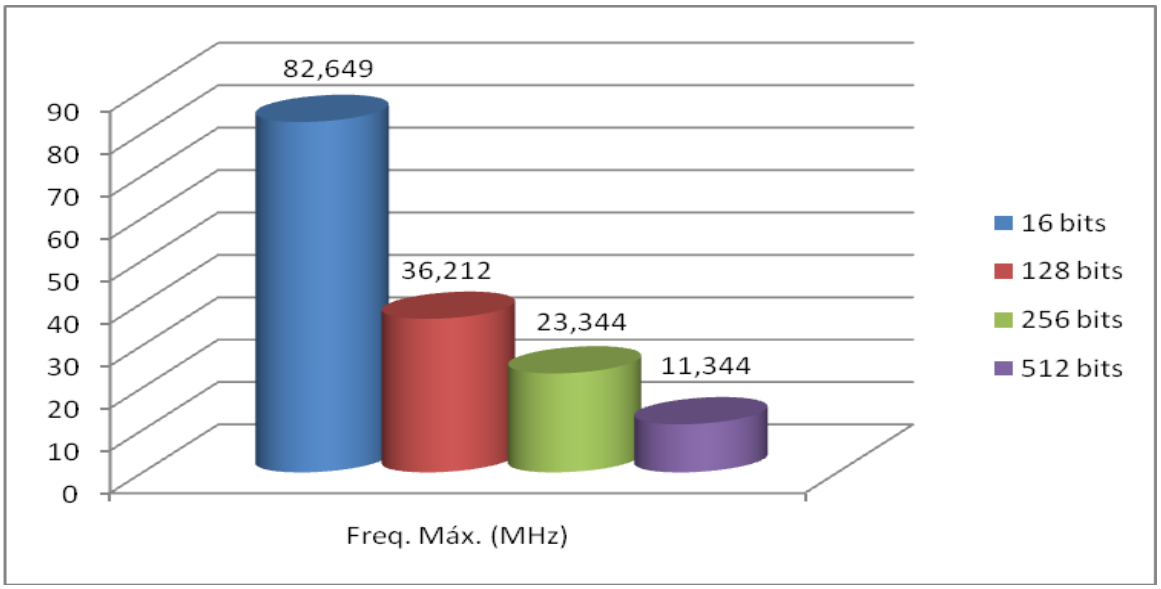

Figura 30.- Gráfica Blum Blum Shub de la frecuencia máxima en función del nivel de seguridad con optimización en Área.

En las siguientes gráficas se mostrarían el conjunto de los resultados obtenidos para la implementación con el esfuerzo alto en tiempo.

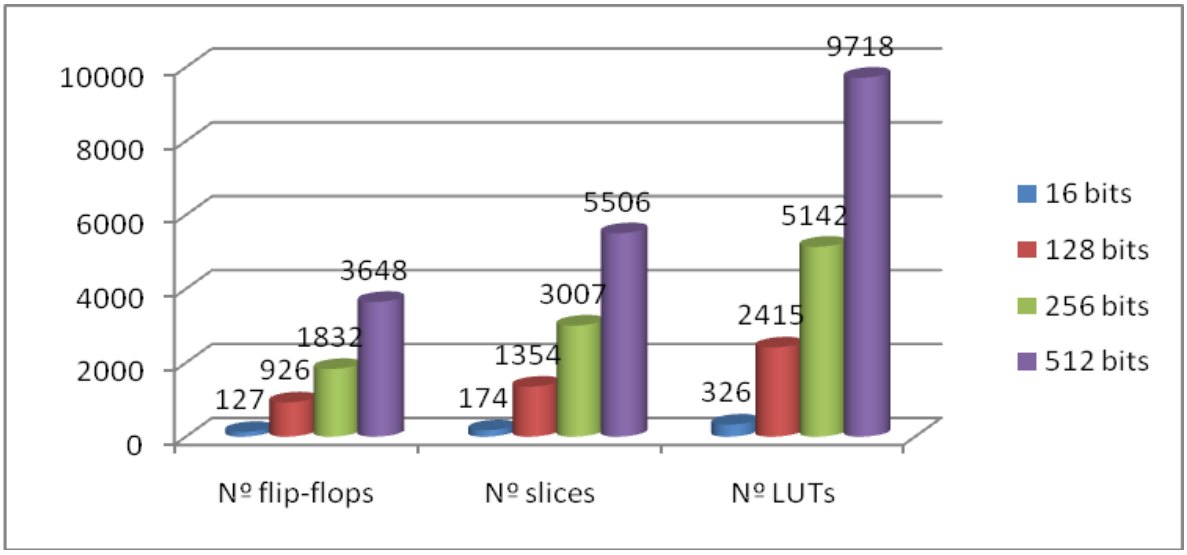

Figura 31.- Gráfica del Blum Blum Shub con el nº de flip-flops, slices y tamaño en LUTs en función del nivel de seguridad con optimización en Tiempo.

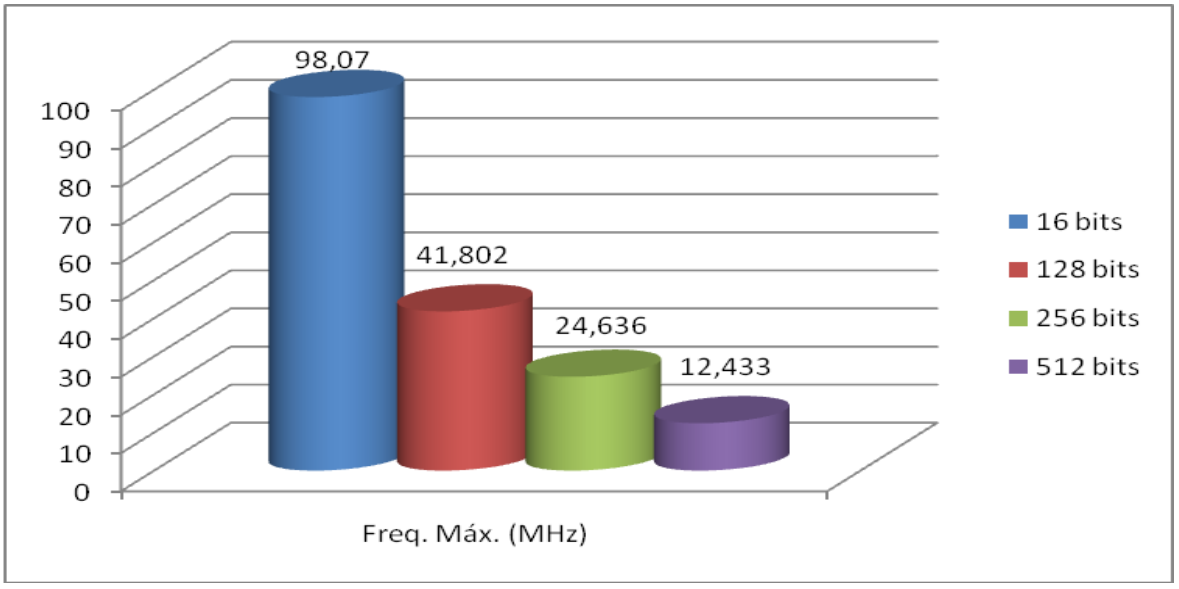

Figura 32.- Gráfica Blum Blum Shub de la frecuencia máxima en función del nivel de seguridad con optimización en Tiempo.

4.1.4.3 Síntesis Synopsys

Se realiza la síntesis con la herramienta Design Vision del programa Synopsys en la que se ha usado la opción de un esfuerzo alto en área a la hora de

realizar esta síntesis, ya que para las especificaciones requeridas en este proyecto es necesario que se ocupe el menor área posible.

En esta síntesis se incluyen los estudios para los diferentes niveles de seguridad de 16, 256 y 512 bits. En esta síntesis se extraen datos tanto de área como de consumo.

De los datos de área se obtendrá el número de *ports* que sería el número de puertos o entradas y salidas que se disponen en el diseño, el número de *nets* que sería el número de uniones o interconexionado, el número de *cells* que son el número total de las celdas o puertas lógicas usadas y el número de *references* que son los distintos tipos de puertas lógicas empleadas en la síntesis de este diseño. Por otra parte se dispondría del área perteneciente a la lógica combinacional y el área perteneciente a la lógica no combinacional además del *Net Interconnect area* que sería la parte de área ocupada por el interconexionado. Finalmente tendremos el *Total cell area* que sería el área total dispuesta para el cómputo total de las puertas lógicas, sumando la parte combinacional y la no combinacional; y si a este valor se le añade el área ocupada por el interconexionado se obtiene el *Total area.*

De los datos referidos al consumo se obtendrá la *Cell Internal Power* que se trata de la potencia consumida internamente por las celdas y la *Net Switching Power* que sería la potencia consumida asociada al interconexionado, que sumando estas dos potencias entre sí, se obtendría el consumo total dinámico para el diseño realizado ó *Total Dynamic Power*. A su vez también se obtendría el *Cell Leakage Power* que serían las pérdidas asociadas a este diseño.

N = 16 bits de nível de seguridad

Datos de Área de la síntesis

```
Number of ports:              53
Number of nets:              268
Number of cells:             208
Number of references:         26

Combinational area:       5081.650954
Noncombinational area:    4985.855022
Net Interconnect area:     575.952228

Total cell area:          10067.505977
Total area:               10643.458205
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   1.2762 mW   (61%)
  Net Switching Power  = 825.6773 uW   (39%)
                         ---------
Total Dynamic Power    =   2.1019 mW  (100%)

Cell Leakage Power     =  35.5641 uW
```

N = 128 bits de nível de seguridad

Datos de Área de la síntesis

```
Number of ports:              389
Number of nets:              1781
Number of cells:             1380
Number of references:          22

Combinational area:       39125.715727
Noncombinational area:    36637.285164
Net Interconnect area:    6073.715567

Total cell area:          75763.000891
Total area:               81836.716457
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   8.3942 mW   (78%)
  Net Switching Power  =   2.4190 mW   (22%)
                         ---------
Total Dynamic Power    =  10.8131 mW  (100%)

Cell Leakage Power     = 274.7039 uW
```

N = 256 bits de nível de seguridad

Datos de Área de la síntesis

```
Number of ports:              773
Number of nets:              3544
Number of cells:             2756
Number of references:          25

Combinational area:       78695.928521
Noncombinational area:    72673.688324
Net Interconnect area:    13847.095226

Total cell area:          151369.616845
Total area:               165216.712071
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =  16.9603 mW   (78%)
  Net Switching Power  =   4.7418 mW   (22%)
                         ---------
Total Dynamic Power    =  21.7020 mW  (100%)

Cell Leakage Power     = 550.3465 uW
```

N = 512 bits de nível de seguridad

Datos de Área de la síntesis

```
Number of ports:              1541
Number of nets:               7064
Number of cells:              5502
Number of references:           25

Combinational area:       158141.799078
Noncombinational area:    144689.355644
Net Interconnect area:    32939.450401

Total cell area:          302831.154722
Total area:               335770.605122
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =  34.0827 mW   (77%)
  Net Switching Power  =  10.1944 mW   (23%)
                         ---------
Total Dynamic Power    =  44.2771 mW  (100%)

Cell Leakage Power     =   1.1242 mW
```

4.1.4.4 Resultados Synopsys

De los datos obtenidos de esta síntesis mediante la herramienta Design Vision del programa Synopsys se obtiene la siguiente tabla con los resultados resumidos de los diferentes diseños sintetizados en función del nível de seguridad empleado:

	16 bits	128 bits	256 bits	512 bits
Ports	53	389	773	1541
Nets	268	1781	3544	7064
Cells	208	1380	2756	5502
References	26	22	25	25
Comb. Area (µm²)	5.082	39.126	78.696	158.142
No comb. Area (µm²)	4.986	36.637	72.674	144.689
Net area (µm²)	576	6.074	13.847	32.939
Total cell area (µm²)	10.068	75.763	151.370	302.831
Total area (µm²)	10.643	81.837	165.217	335.771
Consumo total (mW)	2,10	10,81	21,70	44,28
Pérdidas (µW)	35,56	274,70	550,35	1124,20

A continuación se muestra una gráfica con el número de puertas lógicas empleadas en cada diseño según los bits empleados a la entrada.

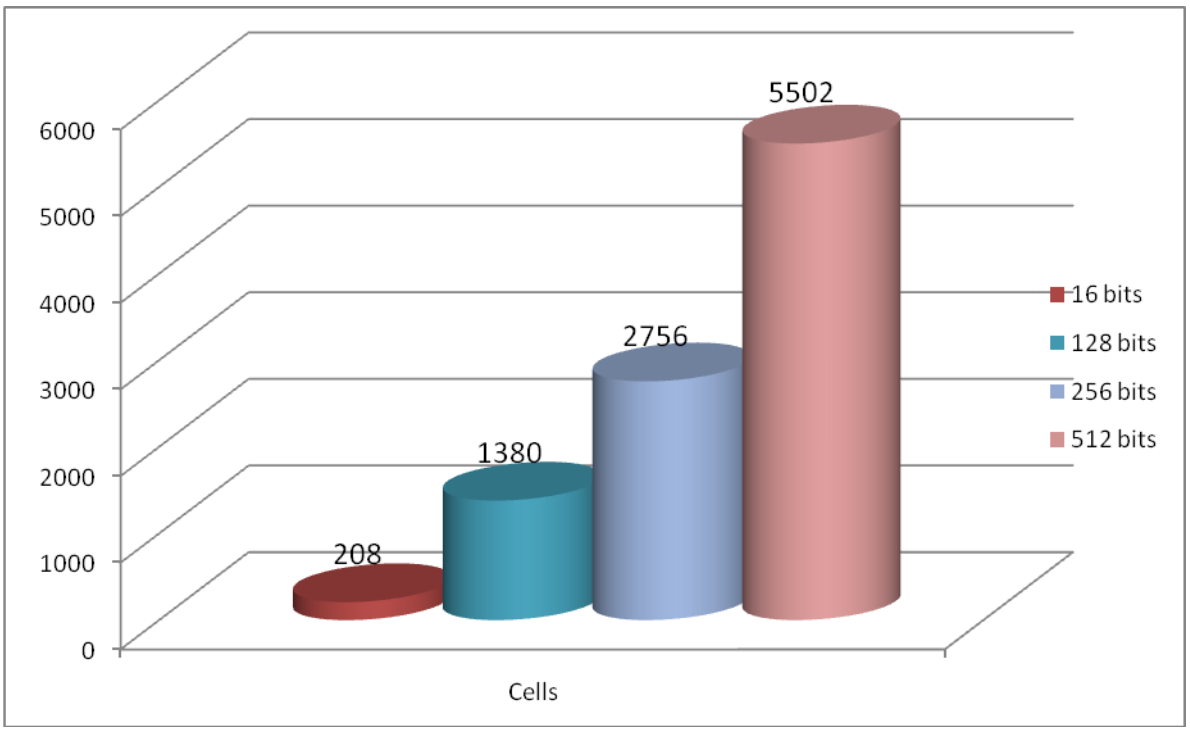

Figura 33.- Gráfica puertas lógicas Blum Blum Shub.

En la siguiente gráfica se muestra el total del área según el número de bits empleados a la entrada con sus respectivas particiones en área correspondiente a la parte de la lógica combinacional y a la parte de la lógica no combinacional.

Cabe destacar que para el total del área a parte de sumar estas dos partes correspondientes a la lógica, habría que añadirle la parte correspondiente al área del interconexionado.

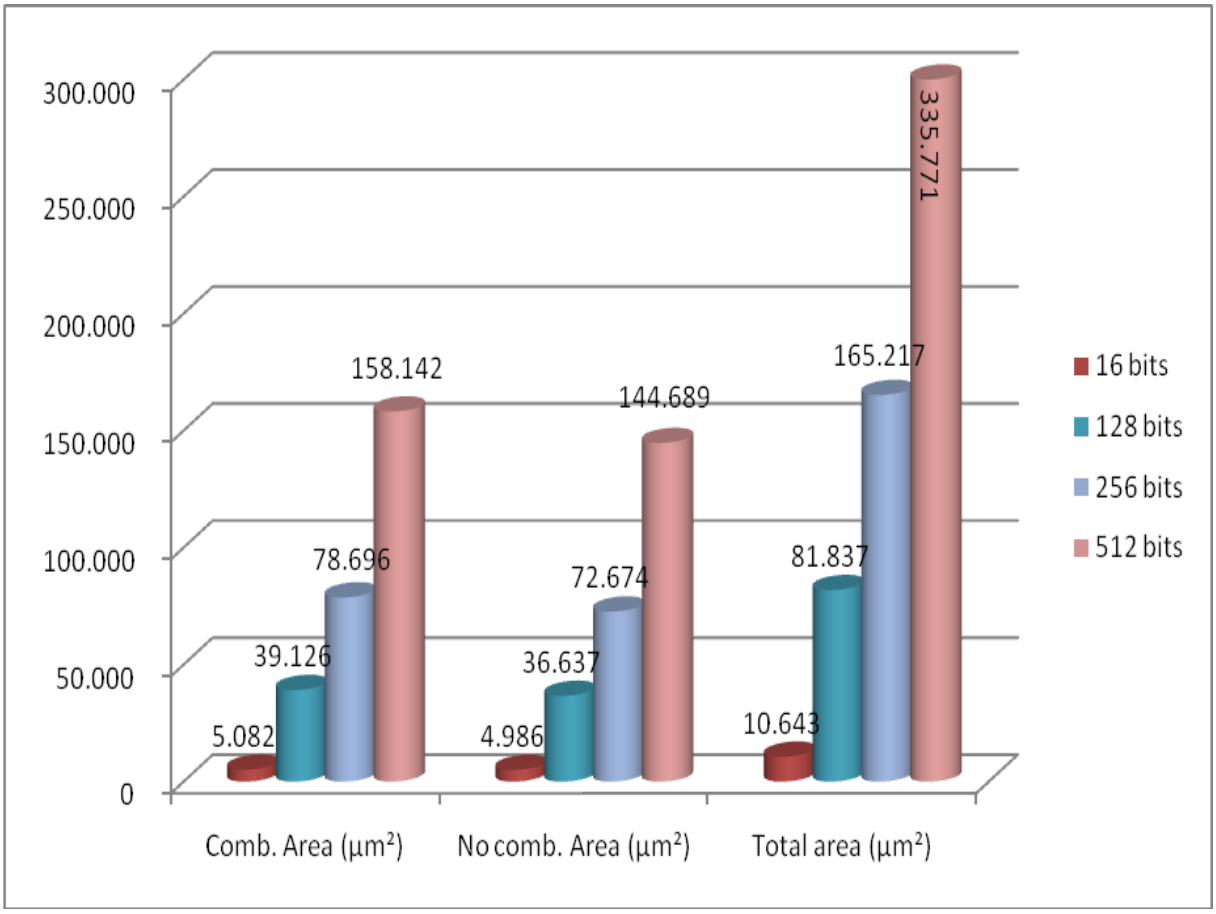

Figura 34.- Gráfica distribución área Blum Blum Shub.

Se muestra la potencia que se consumiría (en unidades de miliwatios) por cada diseño en función del número de bits que se utilicen a la entrada por medio de la siguiente gráfica:

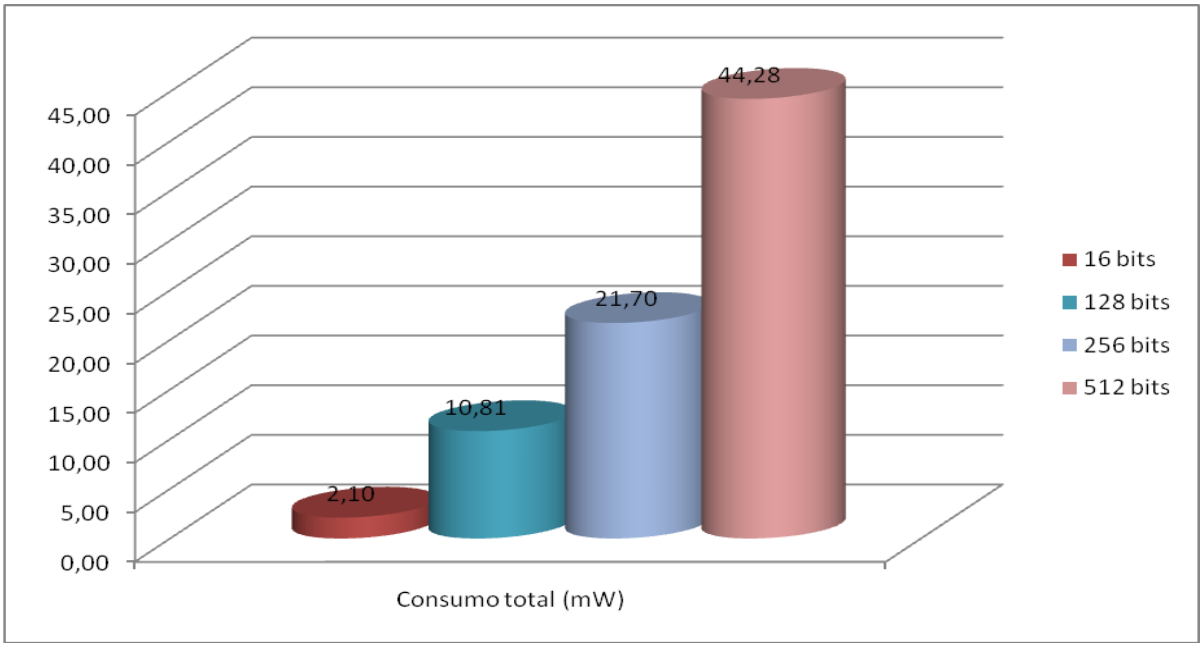

Figura 35.- Gráfica consumo Blum Blum Shub.

4.1.5 Conclusiones

Una vez realizado el diseño de este generador Blum Blum Shub de números pseudo-aleatorios en el lenguaje de descripción hardware vhdl y sus posteriores implementaciones tanto para la tecnología de dispositivos programables FPGAs como para la tecnología de circuitos a medida ASICs, se pueden analizar los resultados obtenidos y llegar a una serie de conclusiones a partir de los mismos.

De los resultados obtenidos en la implementación de dispositivos programables a partir del programa ISE, se puede observar que comparando en la figura 28 (para el caso del diseño con un nivel de seguridad de 128 bits) los dos tipos de optimizaciones llevados a cabo en el caso de la optimización efectuada con un alto esfuerzo en área se consigue reducir ligeramente el tamaño del área medida en LUTs, así como el número de flip-flops y el número de slices respecto a la optimización con un alto esfuerzo en requerimientos de tiempo; sin embargo, con la optimización en tiempo se consigue una frecuencia máxima mayor con lo que el periodo mínimo sería menor y se conseguiría aumentar la velocidad.

Si para esta misma síntesis de dispositivos programables se compara en función del nivel de seguridad ó número de bits a tratar en el proceso, se comprueba que a medida que aumenta el número de bits se va incrementando exponencialmente tanto el tamaño del área como el número de flip-flops y slices (figuras 29 y 31), a cambio, se observa que a medida que se aumenta el número de bits a tratar se reduce sustancialmente la frecuencia máxima haciendo los procesos cada vez más lentos (figuras 30 y 32).

De la síntesis realizada para circuitos integrados a medida también se extrae la conclusión de que a medida que se aumenta el número de bits a tratar se incrementan las puertas lógicas, el interconexionado, el tamaño del área de estas celdas, ya sea en la parte correspondiente a la lógica combinacional como a la parte correspondiente a la lógica no combinacional. Así mismo, el estudio realizado del consumo de la potencia medida en miliwatios nos aporta que se consume bastante más a medida que aumenta el número de bits para estos generadores de números aleatorios, esto mismo ocurre con las pérdidas asociadas a esta potencia aunque en magnitudes mucho más inferiores ya que son del orden de microwatios.

DISEÑO PRNG´S

(PSEUDO-RANDOM NUMBER GENERATION)

OPCIÓN A

ALGORITMO – IMPLEMENTACIÓN VHDL – SIMULACIÓN – COMPROBACIÓN LENGUAJE C – SÍNTESIS ISE Y SYNOPSYS – RESULTADOS – CONCLUSIONES

4.2 Opción A

4.2.1 Algoritmo

En este apartado se lleva a cabo el estudio del primero de los algoritmos de los generadores de números pseudo-aleatorios que se quieren implementar en las FPGA´s, se muestra en el correspondiente código de lenguaje C que se disponía de partida para el cual se habían realizado pruebas previas para observar la calidad de la salida.

Este algoritmo se trata de un PRNG ultraligero que está basado en el uso de una función triangular.

```
Opción A


   x[0]= x[0] + ((x[0]*x[0])|5);
   x[1]= x[1] + ((x[1]*x[1])|13);

//Non-linerar function


   z[0]= x[0];
   for (i=0; i<64; i++){ z[0]= (z[0]>>1)+(z[0]<<1)+z[0]+x[1];}

//Filter output
// z[0] es la salida del generador
// me quedo con los bits menos significativos que son los que varían mas
   rápidamente
// para el ejemplo de variables de 32 bits me quedo con los 16 menos
   significativos


    z[0]= z[0]&0x0000ffff;
```

El estudio para los tres algoritmos en principio era para los siguientes casos en función de los bits de las variables o datos de entrada:

Variables de 8 bits de entrada -> 4 bits de salida
Variables de 16 bits de entrada -> 8 bits de salida
Variables de 32 bits de entrada -> 16 bits de salida
Variables de 64 bits de entrada -> 48 bits de salida

Pero a su vez, para hacerlo aún más exhaustivo también se ha añadido el estudio para *variables de 128, 196, 256, 512 y 1024 bits de entrada*; y como para estos casos no se disponía de información para saber que filtro de salida se le aplicaba, se ha establecido que el filtro de salida sea tal que los *bits de salida sean la mitad* de la dimensión de los bits de las variables de entrada, tal y como se hace en los tres primeros casos de estos cuatro en los que se disponía de la información, quedando de la siguiente manera:

Variables de 128 bits -> 64 bits de salida
Variables de 196 bits -> 98 bits de salida
Variables de 256 bits -> 128 bits de salida
Variables de 512 bits -> 256 bits de salida
Variables de 1024 bits -> 512 bits de salida

El algoritmo A que se muestra en este apartado parte de dos valores *semilla*, x[0] y x[1], que serían las variables de entrada (para el código mostrado como ejemplo de una dimensión de 32 bits) a los cuales se les realiza una serie de operaciones para luego obtener unas constantes que se emplean en una serie de iteraciones.

Las operaciones que se realizan a estos valores *semilla* serían una multiplicación binaria de sí mismos, posteriormente se le aplicaría un OR lógico a nivel de bit (con el valor en binario del número decimal 5 para x[0] y con el número decimal 13 para x[1]); después a ese valor obtenido se le sumaría el valor *semilla*. Finalmente esos valores obtenidos se truncarían para mantener el mismo número de bits (mismas dimensiones) que las variables de entrada y ahora ya dispondríamos de las constantes a utilizar para las iteraciones.

El siguiente paso en este algoritmo A sería el de asignarle a z[0] como valor inicial la constante obtenida de x[0].

Ese valor inicial de z[0] sirve para comenzar una serie de 64 iteraciones, que consiste en la suma total del valor actual de z[0], un desplazamiento de z[0] a la derecha, un desplazamiento de z[0] a la izquierda más la constante x[1]. Esta serie de iteraciones es la que hace posible el encriptamiento de los datos de entrada.

El último paso consistiría en filtrar la salida quedándose únicamente con los bits menos significativos.

*NOTA: Para ver como se usan a nivel de bit los operadores empleados en estos algoritmos se puede consultar el anexo B.

4.2.2 Implementación en código Vhdl

Para la implementación de este algoritmo A en vhdl se ha llevado a cabo mediante la creación de una máquina de estados. Dicha implementación se rige por la siguiente entidad:

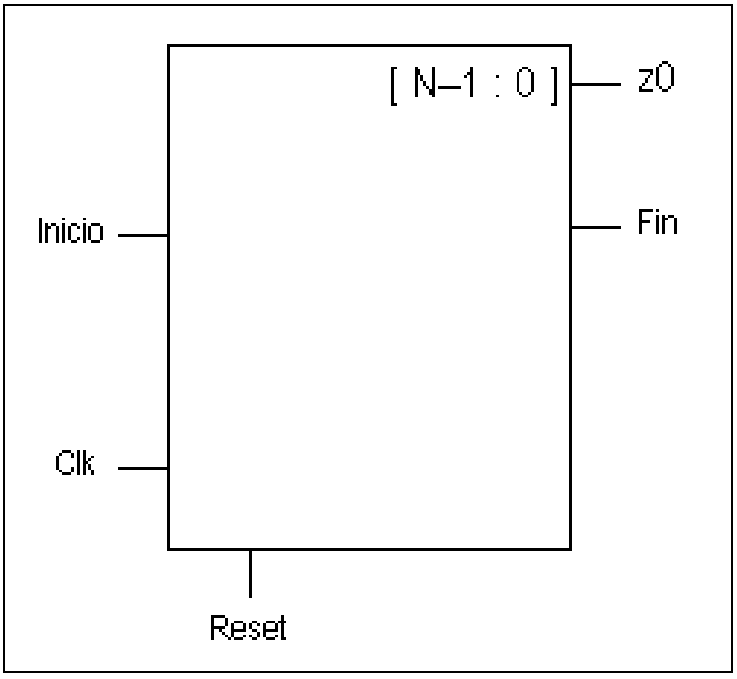

Figura 36.- Entidad algoritmo opción A.

De esta entidad se observa que no hay señales de entrada específicas para la introducción de los datos o variables de entrada, es decir, con los bits que vamos a tratar para generar los números aleatorios.

Estos datos iniciales se introducen como constantes para que así no se sinteticen estos valores que pueden ser aleatorios según queramos, bastaría con

cambiar los números decimales de las constantes que se usan en el código vhdl diseñado. Para probar de mejor manera la funcionalidad de estos diseños que se han creado, se han introducido diferentes valores iniciales como constantes según el número de bits de las variables o datos de entrada.

Ocurre un problema al introducir números muy grandes en vhdl a la hora de realizar la simulación puesto que aparece el siguiente mensaje de error *"Integer value exceeds INTEGER'high"*, esto quiere decir que al simular no soporta números enteros excesivamente elevados ya que el número máximo entero con el que se puede tratar en vhdl es el 2.147.483.647, por lo que para los casos de 64 bits de entrada y mayores hemos mantenido las mismas constantes que para variables de 32 bits de entrada, quedando los valores iniciales que hemos introducido como ejemplos de la siguiente manera, siendo seed_0 el valor inicial (semilla) de x[0] y seed_1 el de x[1], reflejando éstos en números decimales:

Variables de 8 bits -> 55 seed_0 y 77 seed_1
Variables de 16 bits -> 5.555 seed_0 y 7.777 seed_1
Variables de 32 bits -> 55.555.555 seed_0 y 777.777.777 seed_1
Variables de 64 bits -> ´´
Variables de 128 bits -> ´´
Variables de 196 bits -> ´´
Variables de 256 bits -> ´´
Variables de 512 bits -> ´´
Variables de 1024 bits -> ´´

Las dos primeras sentencias del algoritmo A que se encargan de realizar las operaciones ya comentadas previamente, son comunes a los tres algoritmos a implementar:

```
x[0]= x[0] + ((x[0]*x[0])|5);
x[1]= x[1] + ((x[1]*x[1])|13);
```

Estas sentencias también se emplean como constantes introduciendo en x[0] y x[1] los valores que se hayan puesto previamente en seed_0 y seed_1. Hay que tener en cuenta que el resultado de las multiplicaciones proporcionará un valor con el doble de bits que la variable a tratar de partida, por lo que éstos resultados de las operaciones se truncarán ajustándolos al número de bits de las variables de entrada. Los valores de estás constantes calculadas para los valores

iniciales de los ejemplos arriba mencionados quedarían con estos valores en decimal:

Variables de 8 bits -> 12 para x*0* y 122 para x*1*
Variables de 16 bits -> 61.664 para x*0* y 65.326 para x*1*
Variables de 32 bits -> 413.433.136 para x*0* y 206.293.086 para x*1*
Variables de 64 bits -> ´´
Variables de 128 bits -> ´´
Variables de 196 bits -> ´´
Variables de 256 bits -> ´´
Variables de 512 bits -> ´´
Variables de 1024 bits -> ´´

La máquina de estados que rige el funcionamiento de este código implementado en vhdl sería la siguiente:

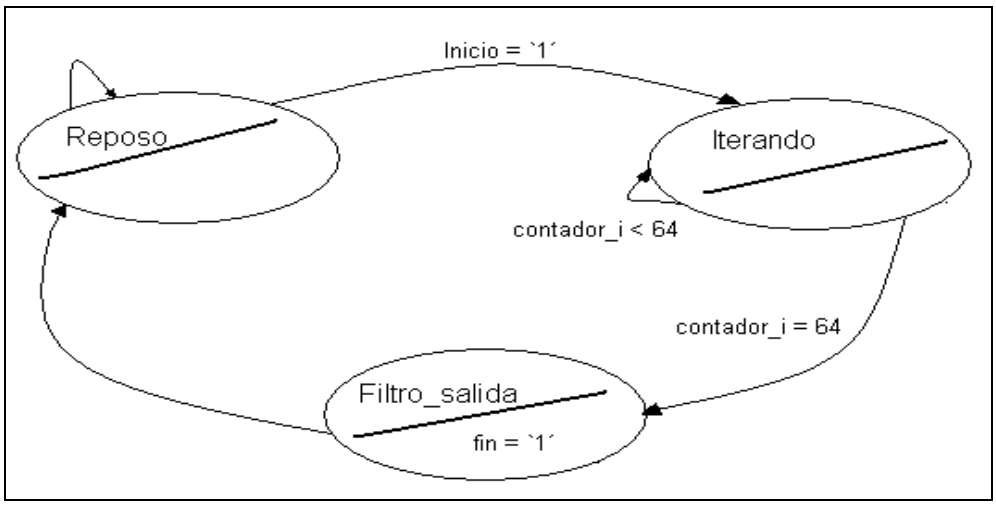

Figura 37.- Diagrama de estados algoritmo opción A.

a) REPOSO: Está a la espera del bit de Inicio para comenzar el proceso.

   Si es 1 go to Iterando, sino mantiene estado

b) ITERANDO: Realiza el mismo proceso durante 64 veces.

   Incrementa contador_i

   Si contador_i = 64 go to Filtro_salida, sino mantiene estado

c) FILTRO_SALIDA: Se queda con los bits menos significativos.

Activa la salida fin

Go to Reposo

El código empleado en la implementación de este algoritmo A se muestra en el apartado 2 del anexo A, en él se muestra el ejemplo para 32 bits de entrada, para los demás casos valdría con cambiar el valor del genérico N que está puesto de tal modo que sea equivalente al nº de bits de salida.

De la arquitectura empleada en esta implementación se obtiene el siguiente esquema de componentes:

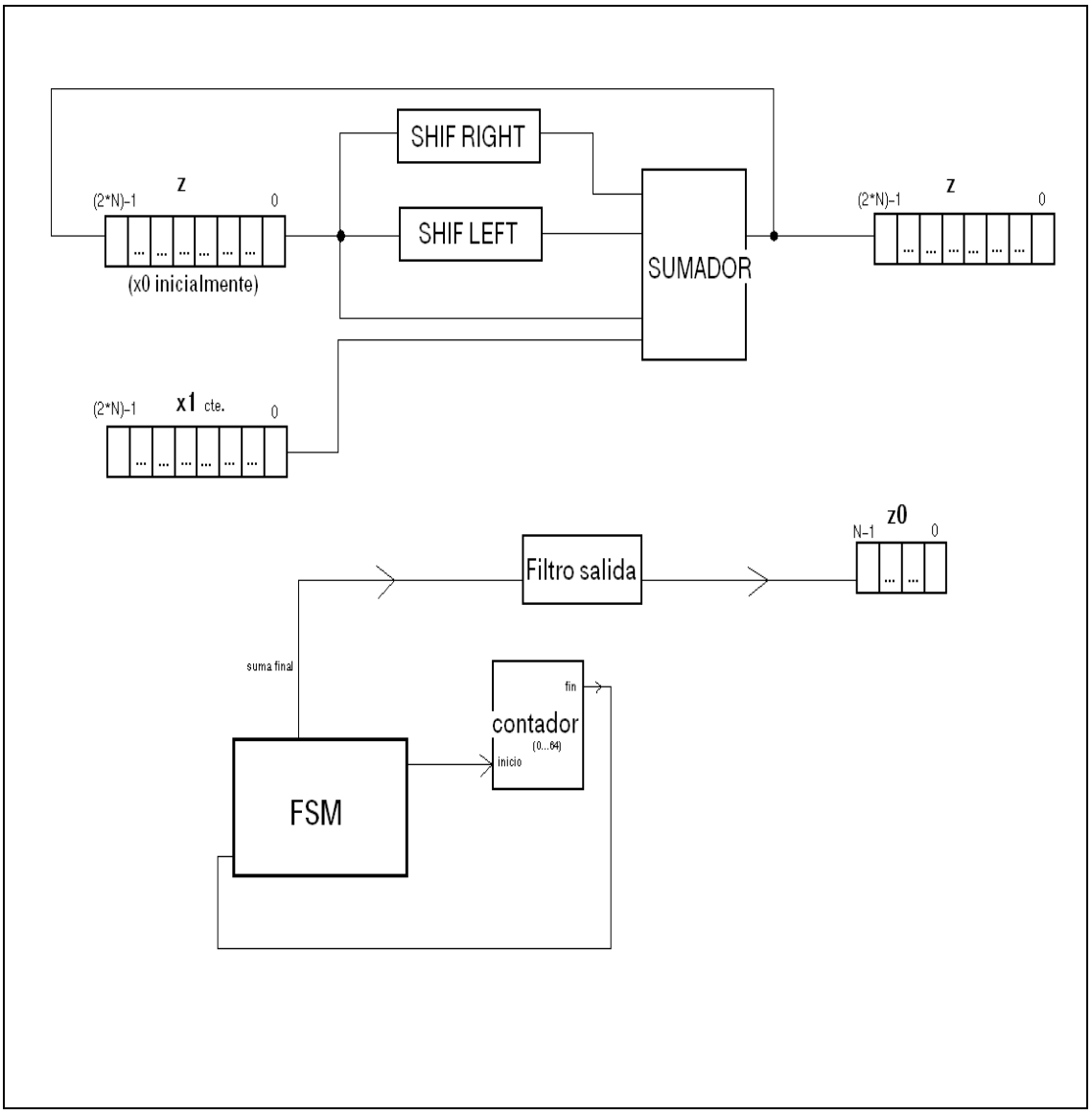

Figura 38.- Esquema de componentes algoritmo opción A.

En él se observa que hay un control (FSM), un contador que va de 0 a 64, un filtro de salida en función del número de bits de salida N, un sumador que tiene por sumandos un registro de desplazamiento a la derecha, un registro desplazamiento a la izquierda a la constante previamente calculada *x1* y a la señal *Z* que inicialmente es *x0* y posteriormente va variando su realimentación.

4.2.3 Simulación

Se procede a la simulación de este algoritmo A, donde en primer lugar se observará a modo de ejemplo la simulación para 32 bits de entrada que genera un número aleatorio en z0.

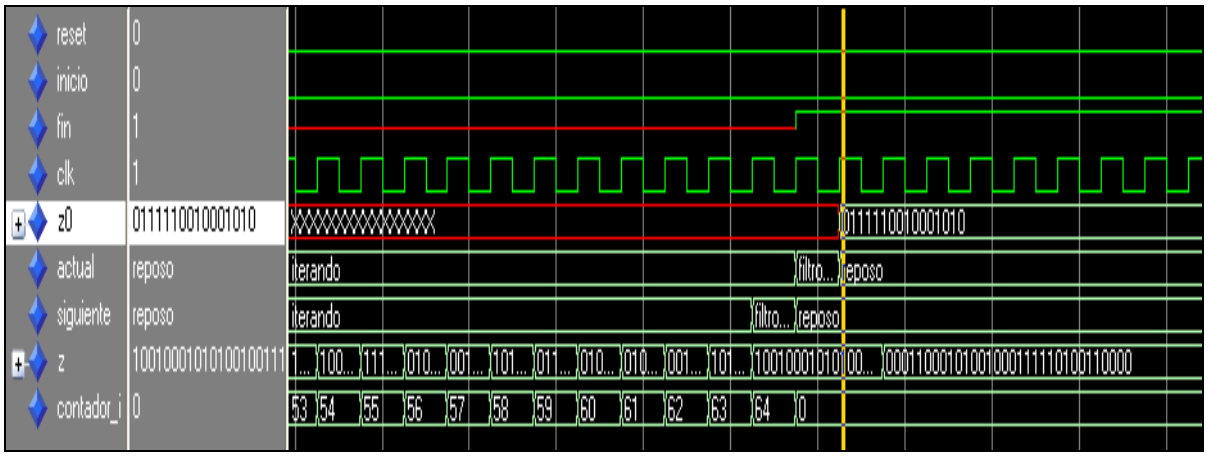

Figura 39.- Simulación 1 algoritmo A.

Se puede observar sencillamente con la salida z0 en binario como con 32 bits de entrada se generan los N = 16 bits de salida.

También se puede observar este resultado en decimal para los 32 bits de entrada.

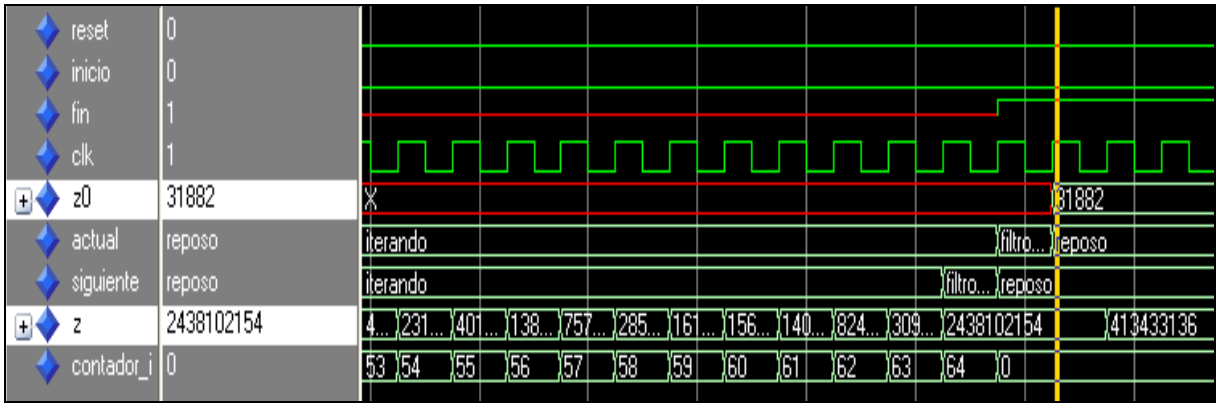

Figura 40.- Simulación 2 algoritmo A.

En la siguiente simulación observamos como opera el algoritmo A que parte del estado en Reposo para proseguir con el proceso de Iteración en el cual va realizando las mismas operaciones sucesivamente mientras el contador_i va incrementándose hasta realizarlas 64 veces.

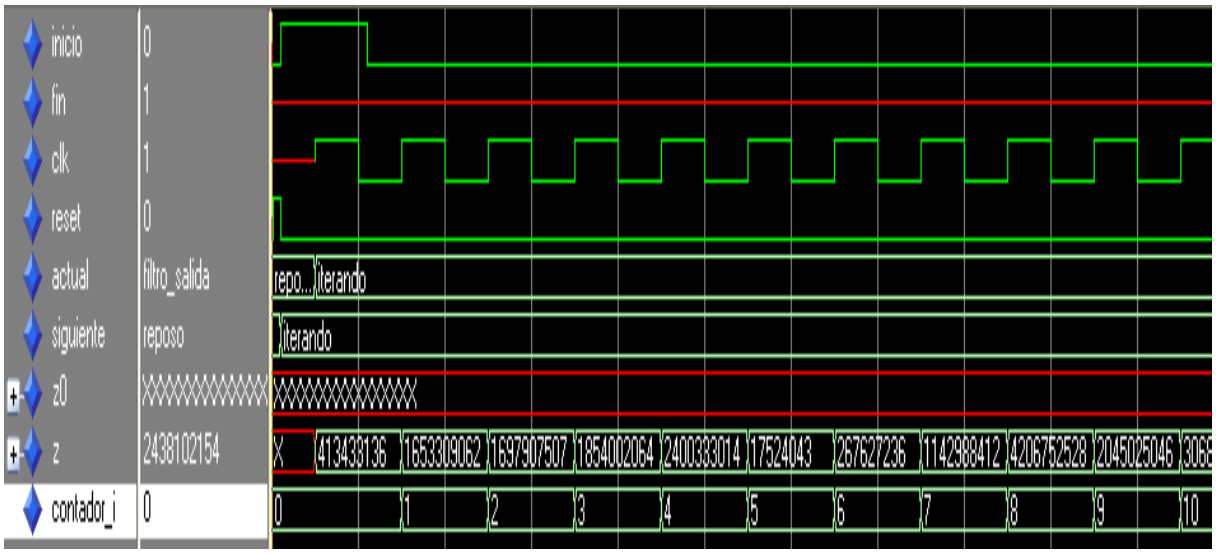

Figura 41.- Simulación 3 algoritmo A.

Se puede destacar como para estos 32 bits de entrada en binario actúa el filtro de salida, despreciando los 16 bits más significativos (a la izquierda de la línea amarilla en z) y quedándose como salida z0 con los 16 bits menos significativos (los de la derecha).

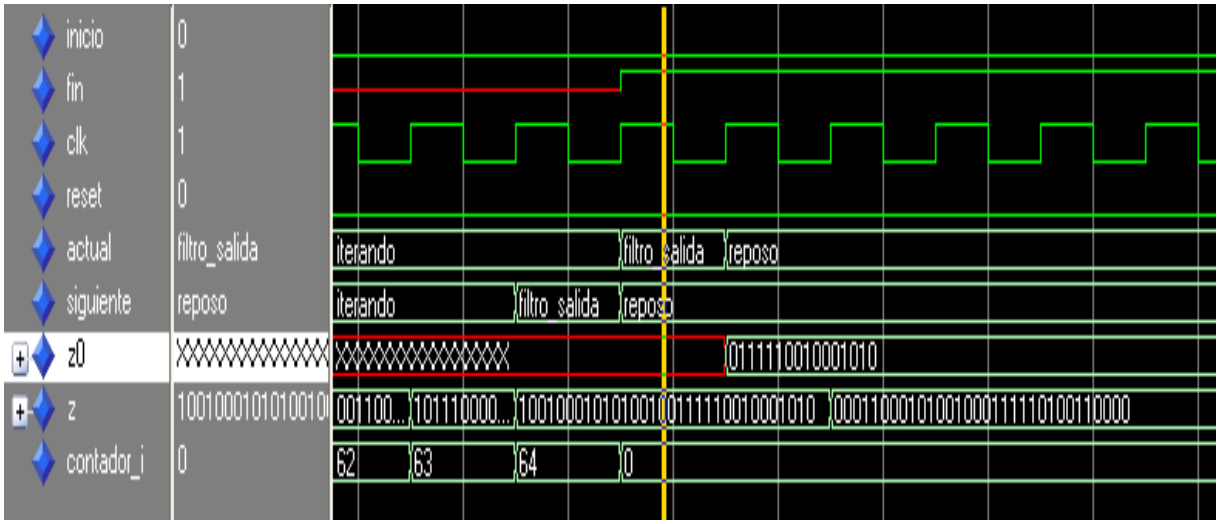

Figura 42.- Simulación 4 algoritmo A.

Se simula para 16 bits de entrada y se observa en decimal los N = 8 bits de salida.

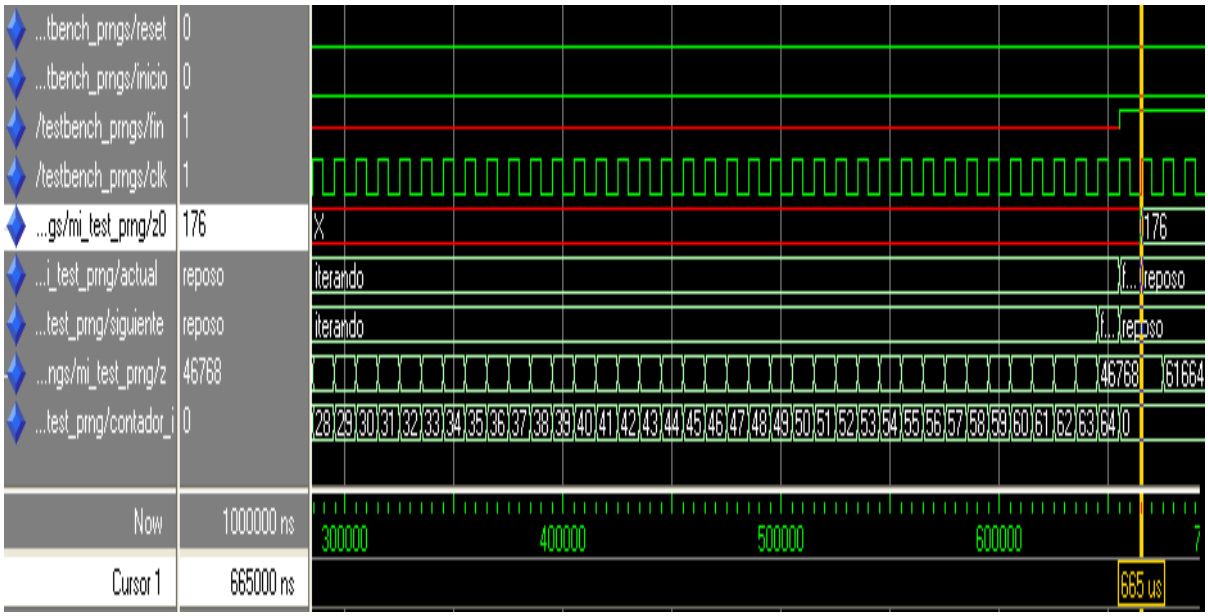

Figura 43.- Simulación 5 algoritmo A.

Lógicamente se ve como para 8 bits de entrada se obtiene una salida z0 más pequeña.

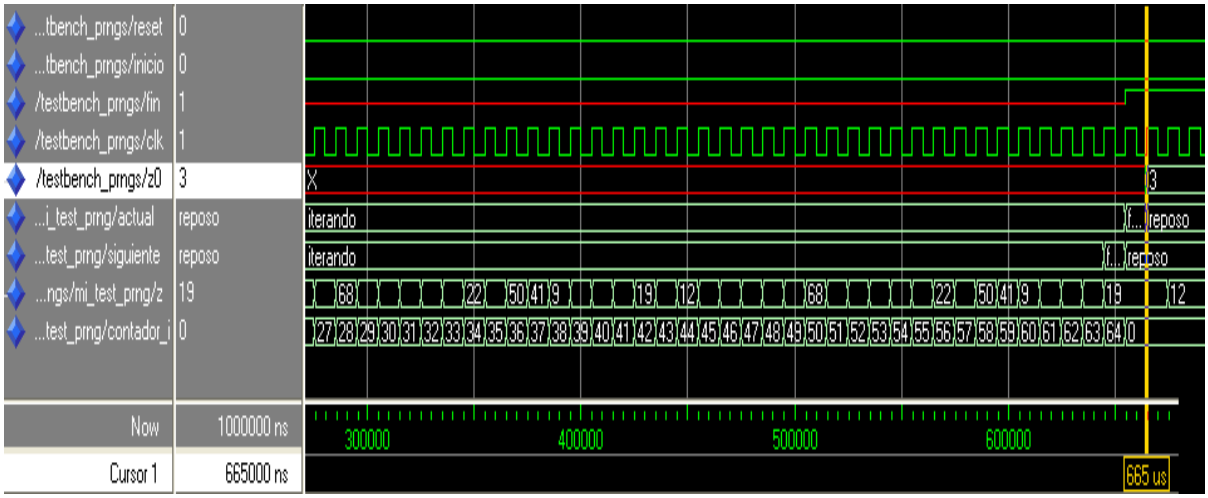

Figura 44.- Simulación 6 algoritmo A.

Simulación para 64 bits de entrada => 48 bits de salida z0.

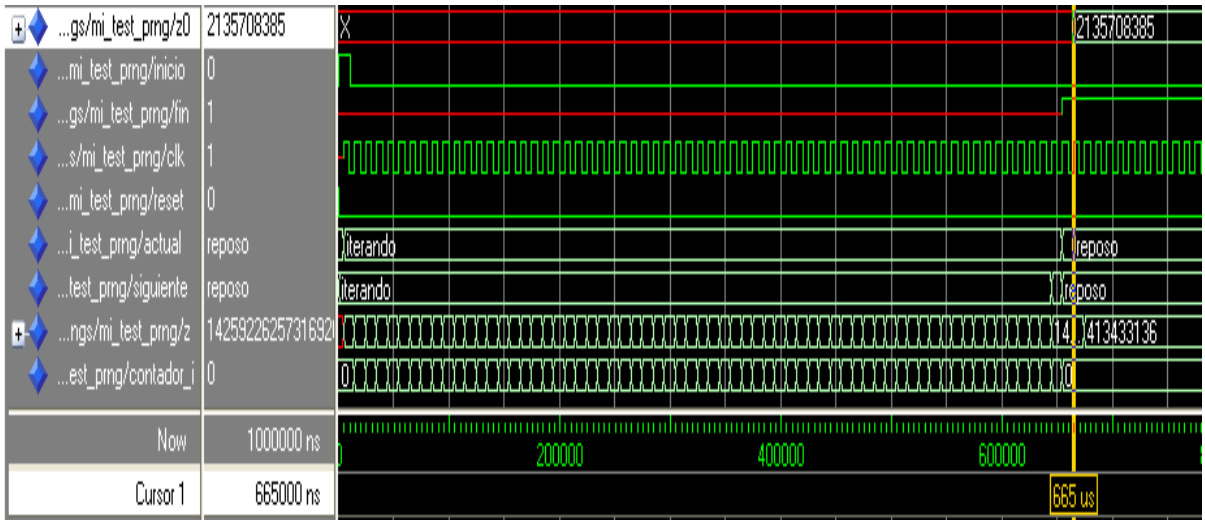

Figura 45.- Simulación 7 algoritmo A.

Ocurre que a medida que aumentan los bits de entrada se obtiene una salida z0 más grande, como para este caso de 128 bits de entrada.

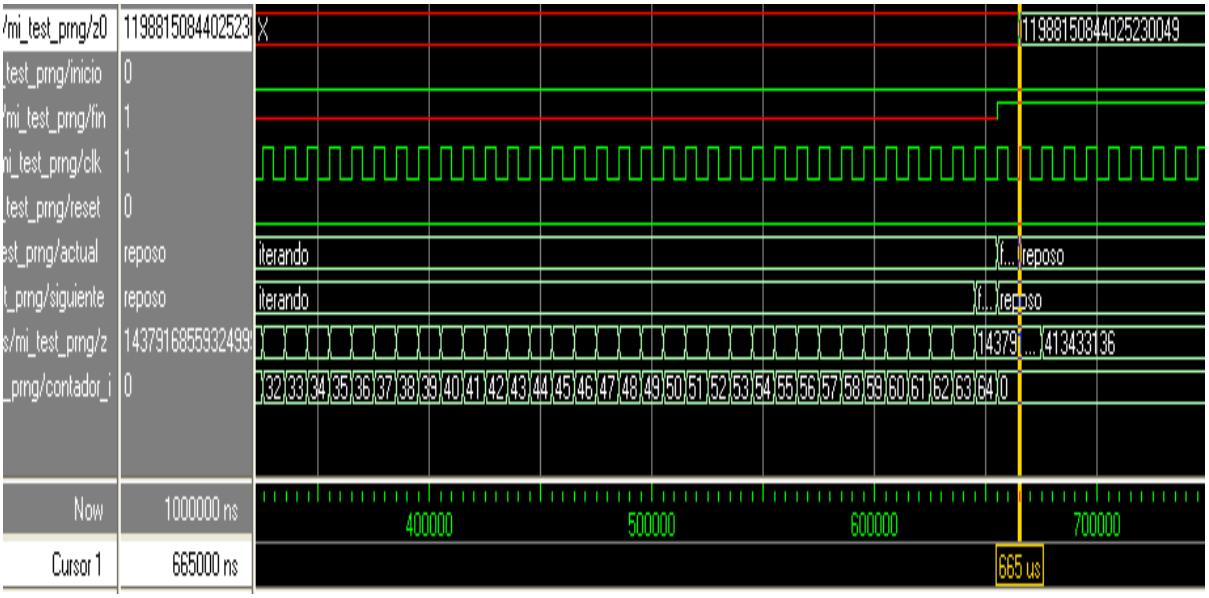

Figura 46.- Simulación 8 algoritmo A.

Simulación para 196 bits de entrada => 98 bits de salida z0.

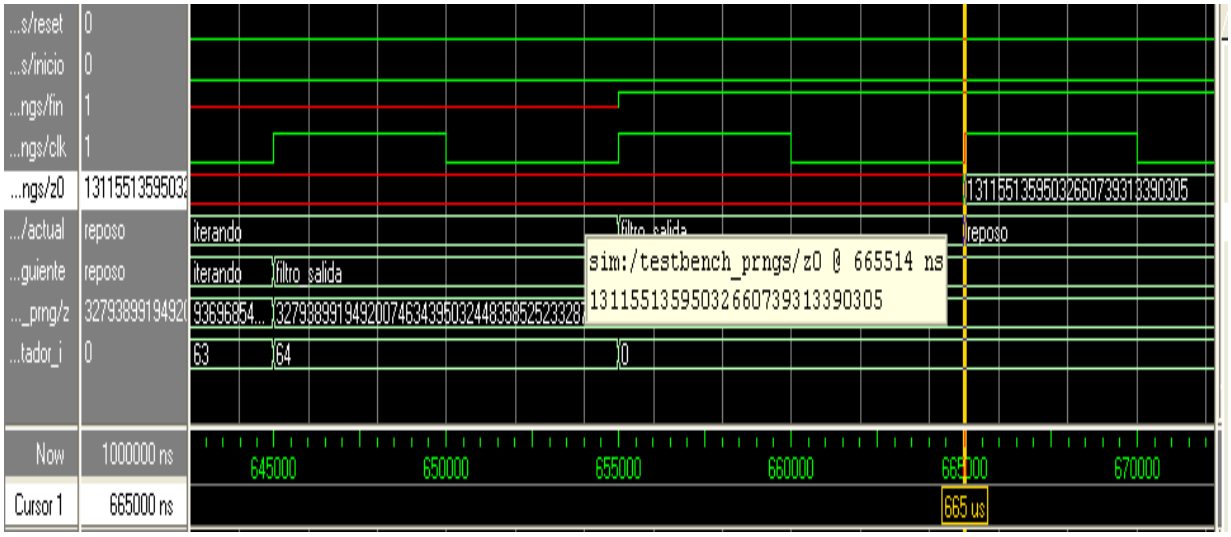

Figura 47.- Simulación 9 algoritmo A.

Simulación para 256 bits de entrada => N = 128 bits de salida z0.

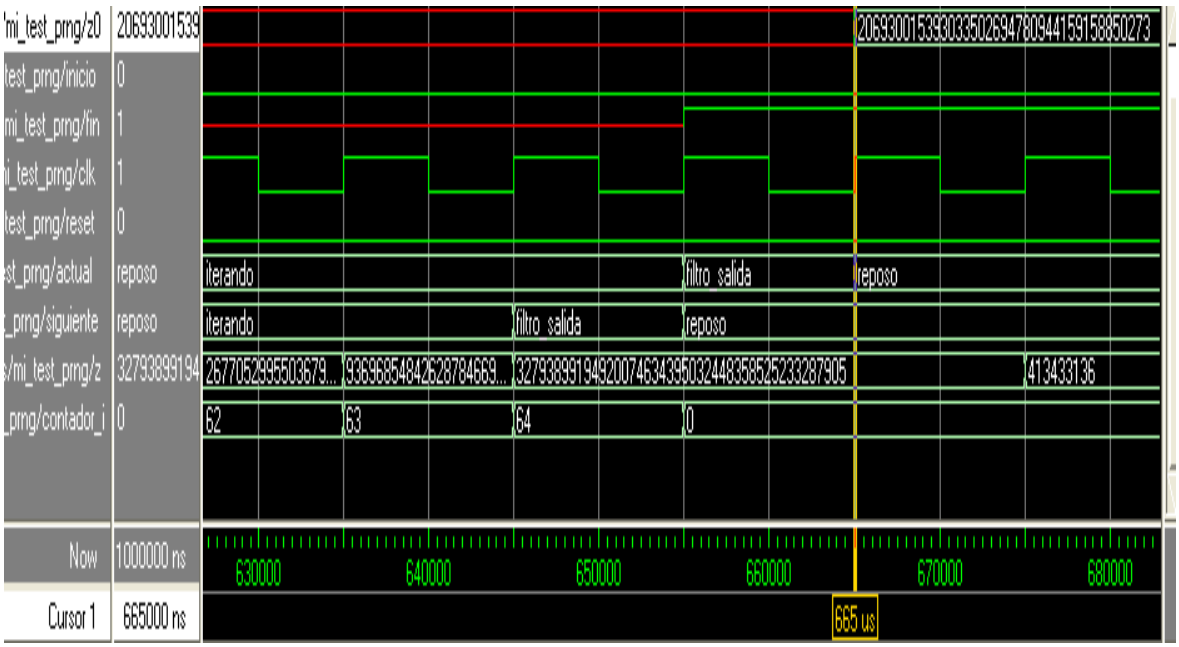

Figura 48.- Simulación 10 algoritmo A.

Simulación para 512 bits de entrada => N = 256 bits de salida z0.

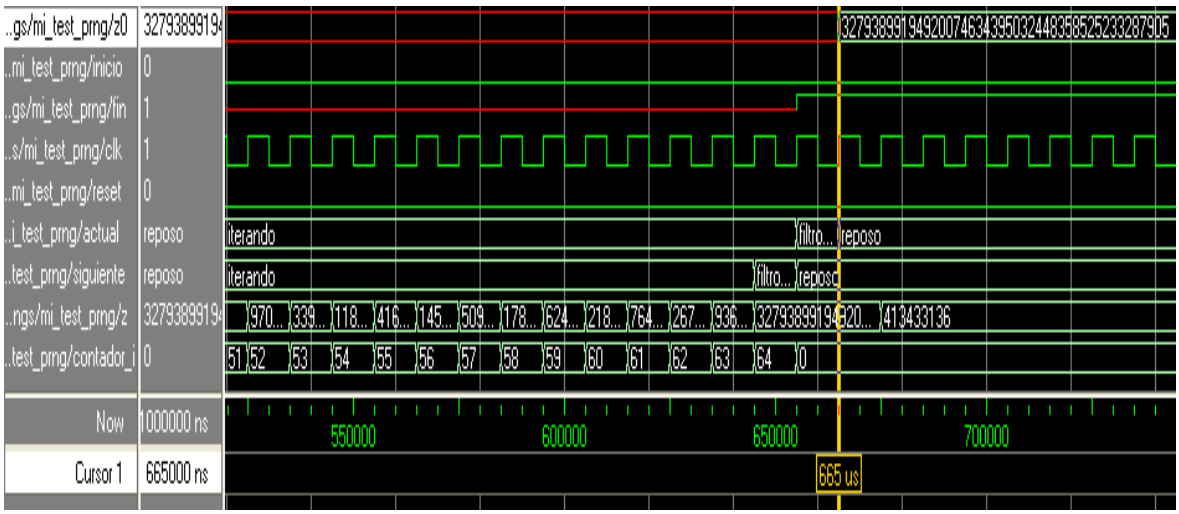

Figura 49.- Simulación 11 algoritmo A.

Simulación de 1024 bits de entrada => N = 512 bits de salida z0.

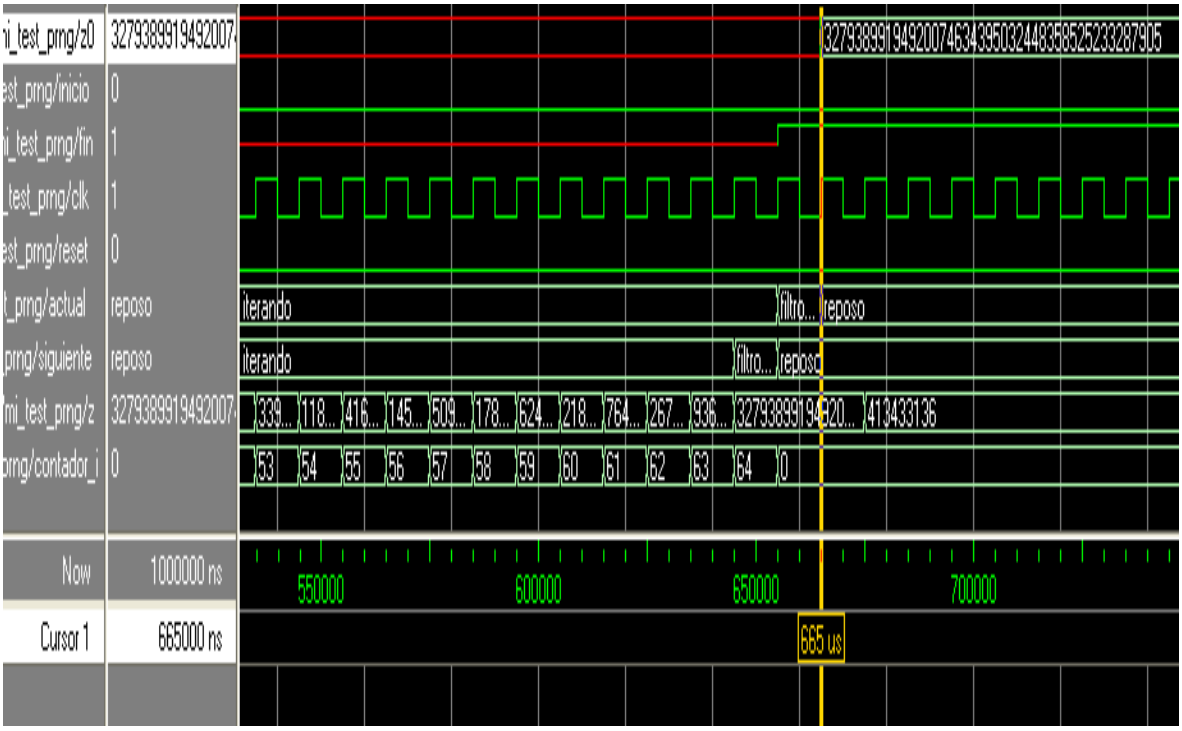

Figura 50.- Simulación 12 algoritmo A.

De estas simulaciones se observa y se comprueba que el algoritmo tarda exactamente el mismo tiempo para ejecutarse independientemente del número de bits que se tenga como entrada de datos o semilla, ya sea para 8 bits como para 1024. Ese tiempo de ejecución es siempre de 665 µs.

A su vez de estas simulaciones se extraen los siguientes resultados como números generados en la salida z0.

Nº de bits de entrada	Genérico N de bits	Nº de bits de la salida z0	Nº generado en la salida z0
8	4	4	3
16	8	8	176
32	16	16	31.882
64	32	48	2.135.708.385
128	64	64	1.198.815.0844.025.230.049
196	98	98	131.155.135.950.326.607.393.133.390.305
256	128	128	206930015393033502694780944159158850273
512	256	256	327938991949200746343950024483585252332879O5
1024	512	512	327938991949200746343950024483585252332879O5

4.2.4 Comprobación lenguaje C

Dada la complejidad de poder comprobar los resultados obtenidos de estos generadores de números pseudo-aleatorios en las simulaciones de Modelsim y el tiempo que conllevaría el operar con la calculadora, además de la facilidad a incurrir en errores a la introducción de datos y operar con números de tan elevada magnitud se ha optado por realizar el código completo y ejecutarlo en Turbo C++.

Se muestra el código para el cual se obtiene el resultado en 16 bits de salida al introducir cualquier número que contenga como máximo 32 bits de entrada.

```
#include <stdio.h>
/* Opcion A. 32 bits de entrada y 16 bits de salida. */
main()
{
 int varcon, i;
unsigned long int x0, x1, z0;
 do {
        printf("\nIntroducir un valor entero para x0: ");
        scanf("%lu", &x0);
        printf("\nIntroducir un valor entero para x1: ");
        scanf("%lu", &x1);
        printf("\nValor inicial de x0: %lu", x0);
        printf("\nValor inicial de x1: %lu", x1);
        x0= x0 + ((x0*x0) | 5);
        x1= x1 + ((x1*x1) | 13);
        printf("\nValor actual de x0: %lu", x0);
        printf("\nValor actual de x1: %lu", x1);
        //Non-linear fuction
```

```
        z0=x0;
        for (i=0; i<64; i++) {
                z0= (z0>>1) + (z0<<1) + z0 + x1;
                }
        //Filter output
        //z0 es la salida del generador,
        //nos quedamos con los bits menos significativos que son los que varian
        //mas rapidamente.
        //para el ejemplo de variables de 16 bits, nos quedamos con los 8 bits
        //menos significativos.
        printf("\nValor decimal de z0: %lu", z0);
        printf("\nValor hexadecimal de z0: %lx", z0);
        z0=z0&0x0000ffff;
        printf("\nValor hexadecimal despues de la mascara de z0: %lx", z0);
        printf("\nValor decimal despues de la mascara de z0: %lu", z0);
        printf("\nPara continuar introducir 1, para parar introducir 0: ");
        scanf("%d", &varcon);
        } while (varcon != 0);
 printf("\nFin del programa");
 }
```

Ahora se muestra el código en el cual se introducen 16 bits de entrada y se obtienen 8 bits de salida.

```
#include <stdio.h>

//Opcion A -- Para 16 bits de entrada y 8 de salida
```

```
main()
{
 unsigned x0, x1, z0, varcon,i;
 do {
        printf("\nIntroducir un valor entero para x0: ");
        scanf("%u", &x0);
        printf("\nIntroducir un valor entero para x1: ");
        scanf("%u", &x1);
        printf("\nValor inicial de x0: %u", x0);
        printf("\nValor inicial de x1: %u", x1);
        x0= x0 + ((x0*x0) | 5);
        x1= x1 + ((x1*x1) | 13);
        printf("\nValor actual de x0: %u", x0);
        printf("\nValor actual de x1: %u", x1);
        //Non-linear fuction
        z0=x0;
for (i=0; i<64; i++) {
                z0= (z0>>1) + (z0<<1) + z0 + x1;
                }
        //Filter output
        //z0 es la salida del generador,
        //nos quedamos con los bits menos significativos que son los que varian
        //mas rapidamente.
        //para el ejemplo de variables de 16 bits, nos quedamos con los 8 bits
        //menos significativos.
        printf("\nValor decimal de z0: %u", z0);
        printf("\nValor hexadecimal de z0: %x", z0);
        z0=z0&0x00ff;
```

```
        printf("\nValor hexadecimal despues de la mascara de z0: %x", z0);
        printf("\nValor decimal despues de la mascara de z0: %u", z0);
        printf("\nPara continuar introducir 1, para parar introducir 0: ");
        scanf("%d", &varcon);
        } while (varcon != 0);
printf("\nFin del programa");
}
```

La diferencia entre estos dos códigos radica en que para la opción de 32 bits de entrada hay que definir a los valores de x0, x1 y z0 como enteros largos y no como enteros como en la opción de 16 bits. Esto es debido al número con que opere el procesador del ordenador, en este caso se realizó en un procesador que opera con 16 bits, por eso se define como *long int* que es de tamaño doble al *int*.

Se comprobó que el resultado de la simulación en Modelsim de la implementación de esta opción A de generador de números pseudo-aleatorios en vhdl concordaba con el resultado obtenido de la ejecución de este código tanto para 32 bits de entrada como para 16.

Se disponen de unos ejecutables[4] para comprobar y obtener cuantos números se quieran con el generador de esta opción A. Para comprobar los resultados obtenidos de los ejecutables con el diseño en vhdl bastaría con cambiar los valores de las constantes seed_0 y seed_1 y calcular los de las constantes x0 y x1.

La comprobación para las opciones mayor número de bits sería redundante además de que no se disponen de procesadores que operen con una cantidad de bits tan elevada.

[4] Estos ejecutables se encuentran en el cd adjunto a este proyecto (su funcionamiento está configurado para procesadores con sistema operativo a 16 bits).

4.2.5 Síntesis

4.2.5.1 Síntesis ISE

De aquí en adelante se muestra la síntesis realizada de este diseño del algoritmo A por medio del programa ISE.

El objetivo es implementar este diseño en una FPGA lo más optima posible en base a nuestros requerimientos, por lo cual debe ser lo más pequeña posible.

Para los casos de menor número de bits a la entrada se ha escogido una FPGA de la familia Spartan 3, más concretamente el dispositivo XC3S50, consiguiendo con éste un gran avance para nuestra tecnología al haberlo conseguido introducir en una FPGA de las más pequeñas del mercado.

Property Name	Value
Product Category	General Purpose
Family	Spartan3
Device	XC3S50
Package	PQ208
Speed	-5

8 bits de entrada -> N = 4 bits de salida

*Implementado con optimización en Área *(Goal: Area, Effort: High)**

Frecuencia máxima = 92,696 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	17	1,536	1%
Number of 4 input LUTs	58	1,536	3%
Logic Distribution			
Number of occupied Slices	31	768	4%
Number of Slices containing only related logic	31	31	100%
Number of Slices containing unrelated logic	0	31	0%
Total Number of 4 input LUTs	58	1,536	3%
Number of bonded IOBs	8	124	6%
IOB Flip Flops	4		
Number of GCLKs	1	8	12%

*Implementado con optimización en Tiempo *(Goal: Speed, Effort: High)**

Frecuencia máxima = 160,648 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	20	1,536	1%
Number of 4 input LUTs	70	1,536	4%
Logic Distribution			
Number of occupied Slices	38	768	4%
Number of Slices containing only related logic	38	38	100%
Number of Slices containing unrelated logic	0	38	0%
Total Number of 4 input LUTs	70	1,536	4%
Number of bonded IOBs	8	124	6%
IOB Flip Flops	4		
Number of GCLKs	1	8	12%

16 bits de entrada -> N = 8 bits de salida

Implementado con optimización en Área

Frecuencia máxima = 116,799 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	25	1,536	1%
Number of 4 input LUTs	77	1,536	5%
Logic Distribution			
Number of occupied Slices	44	768	5%
Number of Slices containing only related logic	44	44	100%
Number of Slices containing unrelated logic	0	44	0%
Total Number of 4 input LUTs	81	1,536	5%
Number used as logic	77		
Number used as a route-thru	4		
Number of bonded IOBs	12	124	9%
IOB Flip Flops	8		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 117,081 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	25	1,536	1%
Number of 4 input LUTs	81	1,536	5%
Logic Distribution			
Number of occupied Slices	45	768	5%
Number of Slices containing only related logic	45	45	100%
Number of Slices containing unrelated logic	0	45	0%
Total Number of 4 input LUTs	85	1,536	5%
Number used as logic	81		
Number used as a route-thru	4		
Number of bonded IOBs	12	124	9%
IOB Flip Flops	8		
Number of GCLKs	1	8	12%

32 bits de entrada -> N = 16 bits de salida

Implementado en Área

Frecuencia máxima = 105,823 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	41	1,536	2%
Number of 4 input LUTs	127	1,536	8%
Logic Distribution			
Number of occupied Slices	75	768	9%
Number of Slices containing only related logic	75	75	100%
Number of Slices containing unrelated logic	0	75	0%
Total Number of 4 input LUTs	145	1,536	9%
Number used as logic	127		
Number used as a route-thru	18		
Number of bonded IOBs	20	124	16%
IOB Flip Flops	16		
Number of GCLKs	1	8	12%

Implementado en Velocidad

Frecuencia máxima = 106,055 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	41	1,536	2%
Number of 4 input LUTs	131	1,536	8%
Logic Distribution			
Number of occupied Slices	76	768	9%
Number of Slices containing only related logic	76	76	100%
Number of Slices containing unrelated logic	0	76	0%
Total Number of 4 input LUTs	149	1,536	9%
Number used as logic	131		
Number used as a route-thru	18		
Number of bonded IOBs	20	124	16%
IOB Flip Flops	16		
Number of GCLKs	1	8	12%

64 bits de entrada -> 48 bits de salida

Implementado con optimización en Área

Frecuencia máxima = 89,081 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	73	1,536	4%
Number of 4 input LUTs	223	1,536	14%
Logic Distribution			
Number of occupied Slices	139	768	18%
Number of Slices containing only related logic	139	139	100%
Number of Slices containing unrelated logic	0	139	0%
Total Number of 4 input LUTs	273	1,536	17%
Number used as logic	223		
Number used as a route-thru	50		
Number of bonded IOBs	36	124	29%
IOB Flip Flops	32		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 89,245 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	73	1,536	4%
Number of 4 input LUTs	225	1,536	14%
Logic Distribution			
Number of occupied Slices	140	768	18%
Number of Slices containing only related logic	140	140	100%
Number of Slices containing unrelated logic	0	140	0%
Total Number of 4 input LUTs	275	1,536	17%
Number used as logic	225		
Number used as a route-thru	50		
Number of bonded IOBs	36	124	29%
IOB Flip Flops	32		
Number of GCLKs	1	8	12%

128 bits de entrada -> N = 64 bits de salida

Implementado con optimización en Área

Frecuencia máxima = 67,669 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	137	1,536	8%
Number of 4 input LUTs	415	1,536	27%
Logic Distribution			
Number of occupied Slices	267	768	34%
Number of Slices containing only related logic	267	267	100%
Number of Slices containing unrelated logic	0	267	0%
Total Number of 4 input LUTs	529	1,536	34%
Number used as logic	415		
Number used as a route-thru	114		
Number of bonded IOBs	68	124	54%
IOB Flip Flops	64		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 67,764 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	137	1,536	8%
Number of 4 input LUTs	417	1,536	27%
Logic Distribution			
Number of occupied Slices	268	768	34%
Number of Slices containing only related logic	268	268	100%
Number of Slices containing unrelated logic	0	268	0%
Total Number of 4 input LUTs	531	1,536	34%
Number used as logic	417		
Number used as a route-thru	114		
Number of bonded IOBs	68	124	54%
IOB Flip Flops	64		
Number of GCLKs	1	8	12%

196 bits de entrada -> N = 98 bits de salida

Implementado con optimización en Área

Frecuencia máxima = 53,903 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	205	1,536	13%
Number of 4 input LUTs	619	1,536	40%
Logic Distribution			
Number of occupied Slices	403	768	52%
Number of Slices containing only related logic	403	403	100%
Number of Slices containing unrelated logic	0	403	0%
Total Number of 4 input LUTs	801	1,536	52%
Number used as logic	619		
Number used as a route-thru	182		
Number of bonded IOBs	102	124	82%
IOB Flip Flops	98		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 53,963 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	205	1,536	13%
Number of 4 input LUTs	621	1,536	40%
Logic Distribution			
Number of occupied Slices	404	768	52%
Number of Slices containing only related logic	404	404	100%
Number of Slices containing unrelated logic	0	404	0%
Total Number of 4 input LUTs	803	1,536	52%
Number used as logic	621		
Number used as a route-thru	182		
Number of bonded IOBs	102	124	82%
IOB Flip Flops	98		
Number of GCLKs	1	8	12%

256 bits de entrada -> N = 128 bits de salida

Para esta implementación de 256 bits de entrada a sido necesario la utilización de una FPGA de la familia Spartan 3 siendo el dispositivo XC3S200 el más pequeño en el que podíamos introducir nuestro diseño para cumplir con nuestros requerimientos de área.

Property Name	Value
Product Category	General Purpose
Family	Spartan3
Device	XC3S200
Package	PQ208
Speed	-5

Implementado con optimización en Área

Frecuencia máxima = 45,7 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	265	3,840	6%
Number of 4 input LUTs	799	3,840	20%
Logic Distribution			
Number of occupied Slices	523	1,920	27%
Number of Slices containing only related logic	523	523	100%
Number of Slices containing unrelated logic	0	523	0%
Total Number of 4 input LUTs	1,041	3,840	27%
Number used as logic	799		
Number used as a route-thru	242		
Number of bonded IOBs	132	141	93%
IOB Flip Flops	128		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 45,743 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	265	3,840	6%
Number of 4 input LUTs	801	3,840	20%
Logic Distribution			
Number of occupied Slices	524	1,920	27%
Number of Slices containing only related logic	524	524	100%
Number of Slices containing unrelated logic	0	524	0%
Total Number of 4 input LUTs	1,043	3,840	27%
Number used as logic	801		
Number used as a route-thru	242		
Number of bonded IOBs	132	141	93%
IOB Flip Flops	128		
Number of GCLKs	1	8	12%

512 bits de entrada -> N = 256 bits de salida

En esta implementación de 512 bits de entrada se ha utilizado el dispositivo XC3S2000 siendo el más pequeño en el que podíamos implementar nuestro diseño.

Property Name	Value
Product Category	General Purpose
Family	Spartan3
Device	XC3S2000
Package	FG456
Speed	-5

Implementado con optimización en Área

Frecuencia máxima = 27,709 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	522	40,960	1%
Number of 4 input LUTs	1,568	40,960	3%
Logic Distribution			
Number of occupied Slices	1,037	20,480	5%
Number of Slices containing only related logic	1,037	1,037	100%
Number of Slices containing unrelated logic	0	1,037	0%
Total Number of 4 input LUTs	2,066	40,960	5%
Number used as logic	1,568		
Number used as a route-thru	498		
Number of bonded IOBs	260	333	78%
IOB Flip Flops	256		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 27,725 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	522	40,960	1%
Number of 4 input LUTs	1,570	40,960	3%
Logic Distribution			
Number of occupied Slices	1,037	20,480	5%
Number of Slices containing only related logic	1,037	1,037	100%
Number of Slices containing unrelated logic	0	1,037	0%
Total Number of 4 input LUTs	2,068	40,960	5%
Number used as logic	1,570		
Number used as a route-thru	498		
Number of bonded IOBs	260	333	78%
IOB Flip Flops	256		
Number of GCLKs	1	8	12%
Total equivalent gate count for design	21,794		

1024 bits de entrada -> N = 512 bits de salida

En esta implementación de 1024 bits de entrada se ha utilizado el dispositivo XC3S5000 siendo el más pequeño en el que podíamos implementar nuestro diseño.

Property Name	Value
Product Category	General Purpose
Family	Spartan3
Device	XC3S5000
Package	FG900
Speed	-5

Implementado con optimización en Área

Frecuencia máxima = 15,502 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,035	66,560	1%
Number of 4 input LUTs	3,106	66,560	4%
Logic Distribution			
Number of occupied Slices	2,064	33,280	6%
Number of Slices containing only related logic	2,064	2,064	100%
Number of Slices containing unrelated logic	0	2,064	0%
Total Number of 4 input LUTs	4,116	66,560	6%
Number used as logic	3,106		
Number used as a route-thru	1,010		
Number of bonded IOBs	516	633	81%
IOB Flip Flops	512		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 15,507 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,035	66,560	1%
Number of 4 input LUTs	3,108	66,560	4%
Logic Distribution			
Number of occupied Slices	2,064	33,280	6%
Number of Slices containing only related logic	2,064	2,064	100%
Number of Slices containing unrelated logic	0	2,064	0%
Total Number of 4 input LUTs	4,118	66,560	6%
Number used as logic	3,108		
Number used as a route-thru	1,010		
Number of bonded IOBs	516	633	81%
IOB Flip Flops	512		
Number of GCLKs	1	8	12%

4.2.5.2 Resultados ISE

De esta síntesis realizada en el programa ISE para el algoritmo A se obtiene los siguientes resultados resumidos en la siguiente tabla en función de los parámetros que son de interés para este estudio (flips-flops, slices, tamaño área (LUTs) y frecuencia máxima).

Primeramente se muestran los datos obtenidos para la optimización en Area.

	8 bits	16 bits	32 bits	64 bits	128 bits
Nº flip-flops	17	25	41	73	137
Nº slices	31	44	75	139	267
Nº LUTs	58	81	145	273	529
Freq. Máx. (MHz)	92,696	116,799	105,823	89,081	67,669

	196 bits	256 bits	512 bits	1024 bits
Nº flip-flops	205	265	522	1035
Nº slices	403	523	1037	2064
Nº LUTs	801	1041	2066	4116
Freq. Máx. (MHz)	53,903	45,7	27,709	15,502

De estos datos se obtiene la siguiente gráfica resumen.

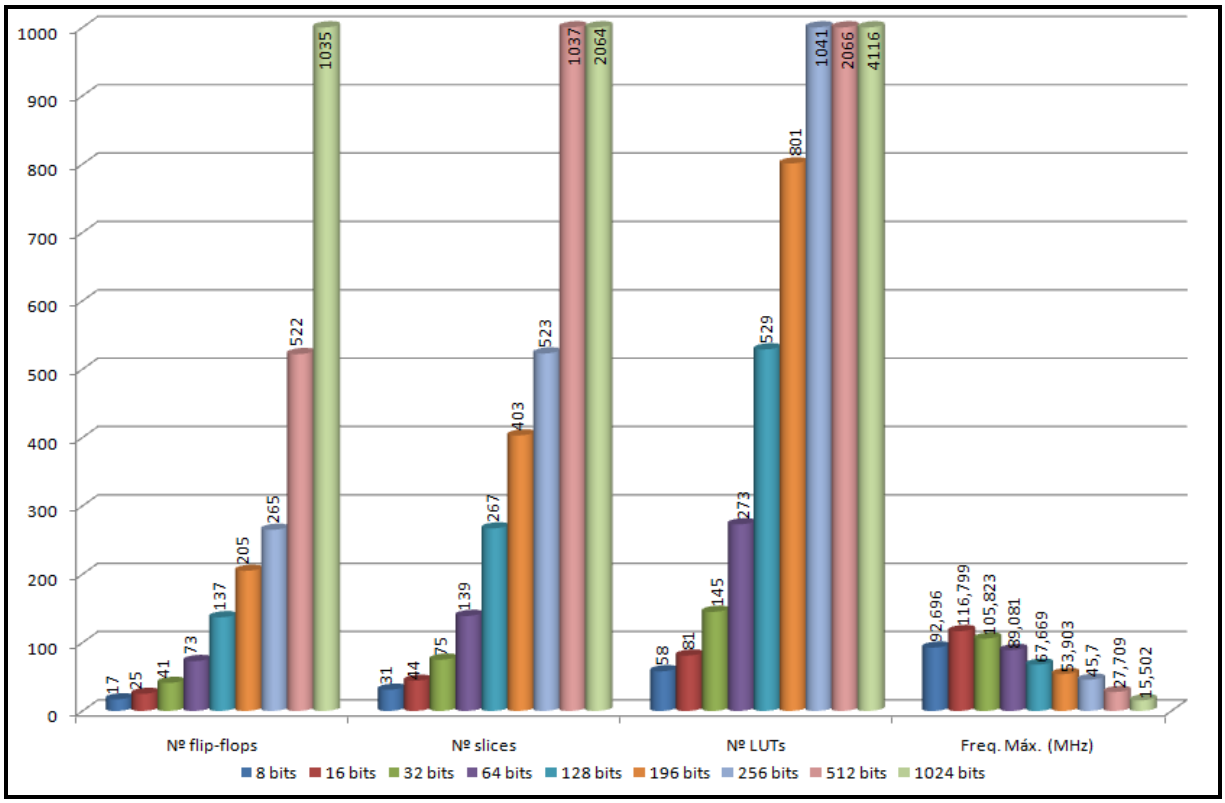

Figura 51.- Gráfica resumen síntesis ISE con optimización en Área del algoritmo A.

En segundo lugar se muestra la tabla con los datos obtenidos para la optimización en Tiempo.

	8 bits	16 bits	32 bits	64 bits	128 bits
Nº flip-flops	20	25	41	73	137
Nº slices	38	45	76	140	268
Nº LUTs	70	85	149	275	531
Freq. Máx. (MHz)	160,648	117,081	106,055	89,245	67,764

	196 bits	256 bits	512 bits	1024 bits
Nº flip-flops	205	265	522	1035
Nº slices	404	524	1037	2064
Nº LUTs	803	1043	2068	4118
Freq. Máx. (MHz)	53,963	45,743	27,725	15,507

De estos nuevos datos se obtiene la siguiente gráfica resumen.

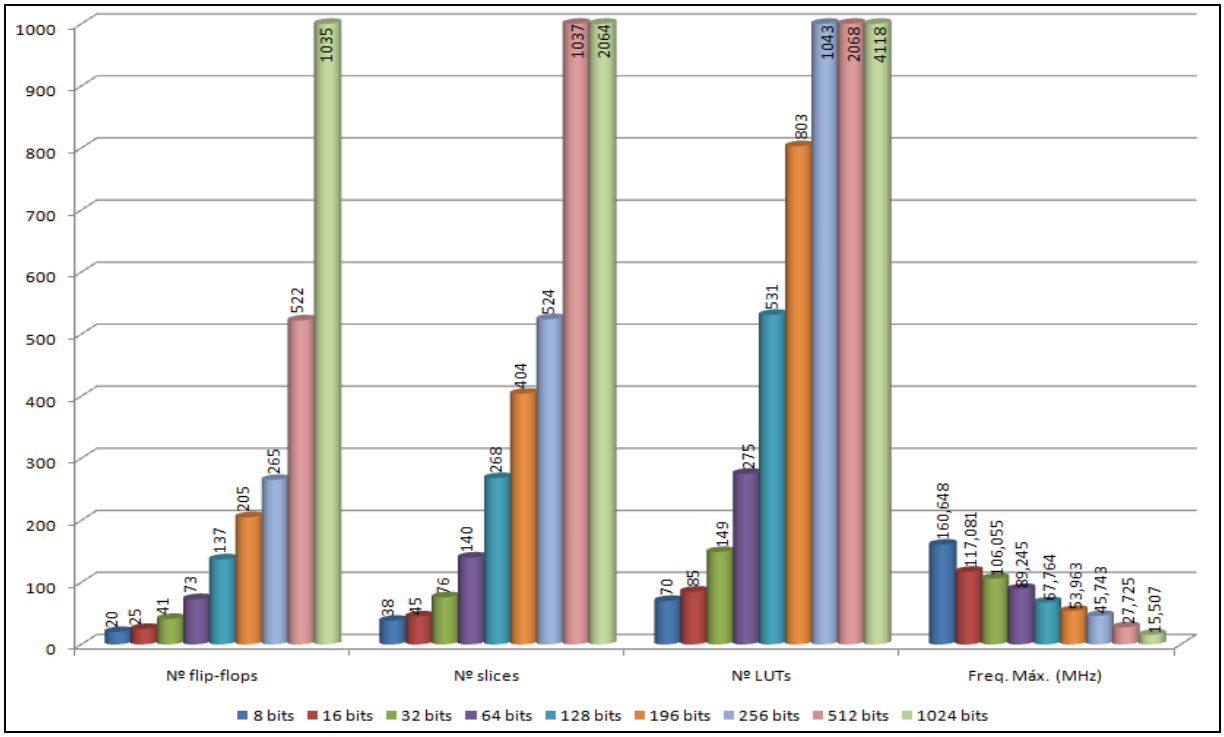

Figura 52.- Gráfica resumen síntesis ISE con optimización en Tiempo del algoritmo A.

4.2.5.3 Síntesis SYNOPSYS

Se realiza la síntesis con la herramienta Design Vision del programa Synopsys en la que se ha usado la opción de un esfuerzo alto en área a la hora de realizar esta síntesis, ya que para las especificaciones requeridas en este proyecto es necesario que se ocupe el menor área posible.

Para esta síntesis del algoritmo A se realiza también el estudio para los casos de 8, 16, 32, 64, 128, 196, 256, 512 y 1024 bits de entrada. En esta síntesis se extraen datos tanto de área como de consumo.

De los datos de área se obtendrá el número de *ports* que sería el número de puertos o entradas y salidas que se disponen en el diseño, el número de *nets* que sería el número de uniones o interconexionado, el número de *cells* que son

el número total de las celdas o puertas lógicas usadas y el número de *references* que son los distintos tipos de puertas lógicas empleadas en la síntesis de este diseño. Por otra parte se dispondría del área perteneciente a la lógica combinacional y el área perteneciente a la lógica no combinacional además del *Net Interconnect area* que sería la parte de área ocupada por el interconexionado. Finalmente tendremos el *Total cell area* que sería el área total dispuesta para el cómputo total de las puertas lógicas, sumando la parte combinacional y la no combinacional; y si a este valor se le añade el área ocupada por el interconexionado se obtiene el *Total area.*

De los datos referidos al consumo se obtendrá la *Cell Internal Power* que se trata de la potencia consumida internamente por las celdas y la *Net Switching Power* que sería la potencia consumida asociada al interconexionado, que sumando estas dos potencias entre sí, se obtendría el consumo total dinámico para el diseño realizado ó *Total Dynamic Power*. A su vez también se obtendría el *Cell Leakage Power* que serían las pérdidas asociadas a este diseño.

8 bits de entrada -> N = 4 bits de salida

Datos de Área de la síntesis

```
Number of ports:                8
Number of nets:               122
Number of cells:               91
Number of references:          24

Combinational area:       1043.281991
Noncombinational area:     588.899998
Net Interconnect area:      55.663444

Total cell area:          1632.181989
Total area:               1687.845432
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =  77.4109 uW   (69%)
  Net Switching Power  =  35.4459 uW   (31%)
                         ---------
Total Dynamic Power    = 112.8568 uW  (100%)

Cell Leakage Power     =   6.2840 uW
```

16 bits de entrada -> N = 8 bits de salida

Datos de Área de la síntesis

```
Number of ports:              12
Number of nets:              204
Number of cells:             160
Number of references:         28

Combinational area:       1929.600986
Noncombinational area:     887.495991
Net Interconnect area:     103.939923

Total cell area:          2817.096976
Total area:               2921.036899
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  = 140.6375 uW   (67%)
  Net Switching Power  =  70.3955 uW   (33%)
                         ---------
Total Dynamic Power    = 211.0330 uW  (100%)

Cell Leakage Power     =  10.8016 uW
```

32 bits de entrada -> N = 16 bits de salida

Datos de Área de la síntesis

```
Number of ports:              20
Number of nets:              258
Number of cells:             144
Number of references:         22

Combinational area:       3701.112926
Noncombinational area:    1484.687977
Net Interconnect area:     158.831366

Total cell area:          5185.800903
Total area:               5344.632270
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  = 248.2227 uW   (64%)
  Net Switching Power  = 140.1629 uW   (36%)
                         ---------
Total Dynamic Power    = 388.3857 uW  (100%)

Cell Leakage Power     =  19.4785 uW
```

64 bits de entrada -> N = 48 bits de salida

Datos de Área de la síntesis

```
Number of ports:              36
Number of nets:              458
Number of cells:             248
Number of references:         23

Combinational area:       7183.505848
Noncombinational area:    2679.071949
Net Interconnect area:     353.851855

Total cell area:          9862.577797
Total area:               10216.429652
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  = 515.1262 uW   (49%)
  Net Switching Power  = 546.4007 uW   (51%)
                         ---------
Total Dynamic Power    =   1.0615 mW  (100%)

Cell Leakage Power     =  35.9841 uW
```

128 bits de entrada -> N = 64 bits de salida

Datos de Área de la síntesis

```
Number of ports:              68
Number of nets:              842
Number of cells:             440
Number of references:         25

Combinational area:       13980.592698
Noncombinational area:    5067.839893
Net Interconnect area:     912.724179

Total cell area:          19048.432591
Total area:               19961.156770
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  = 719.0715 uW   (50%)
  Net Switching Power  = 730.8573 uW   (50%)
                         ---------
Total Dynamic Power    =   1.4499 mW  (100%)

Cell Leakage Power     =  66.9025 uW
```

196 bits de entrada -> N = 98 bits de salida

Datos de Área de la síntesis

```
Number of ports:              102
Number of nets:              1331
Number of cells:              725
Number of references:          23

Combinational area:       21588.401556
Noncombinational area:    7605.905834
Net Interconnect area:    1152.505627

Total cell area:          29194.307390
Total area:               30346.813017
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   1.0543 mW   (72%)
  Net Switching Power  = 419.7809 uW   (28%)
                         ---------
Total Dynamic Power    =   1.4740 mW  (100%)

Cell Leakage Power     = 103.8451 uW
```

256 bits de entrada -> N = 128 bits de salida

Datos de Área de la síntesis

```
Number of ports:              132
Number of nets:              1745
Number of cells:              959
Number of references:          21

Combinational area:       28211.932424
Noncombinational area:    9845.375782
Net Interconnect area:    1623.921024

Total cell area:          38057.308206
Total area:               39681.229229
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   1.2401 mW   (78%)
  Net Switching Power  = 350.0031 uW   (22%)
                         ---------
```

```
Total Dynamic Power    =   1.5901 mW  (100%)

Cell Leakage Power     = 135.4751 uW
```

512 bits de entrada -> N = 256 bits de salida

Datos de Área de la síntesis

```
Number of ports:              260
Number of nets:              3461
Number of cells:             1907
Number of references:          21

Combinational area:       56344.622852
Noncombinational area:    19400.447559
Net Interconnect area:    3742.740388

Total cell area:          75745.070412
Total area:               79487.810800
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   2.4302 mW   (77%)
  Net Switching Power  = 717.0370 uW   (23%)
                         ---------
Total Dynamic Power    =   3.1472 mW  (100%)

Cell Leakage Power     = 269.4922 uW
```

1024 bits de entrada -> N = 512 bits de salida

Datos de Área de la síntesis

```
Number of ports:              516
Number of nets:              6916
Number of cells:             3826
Number of references:          21

Combinational area:      112726.163714
Noncombinational area:    38510.591114
Net Interconnect area:    8835.893829

Total cell area:         151236.754828
Total area:              160072.648657
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   4.9187 mW   (77%)
  Net Switching Power  =   1.4949 mW   (23%)
                         ---------
Total Dynamic Power    =   6.4136 mW  (100%)

Cell Leakage Power     = 538.4010 uW
```

4.2.5.4 Resultados SYNOPSYS

En la siguiente tabla resumen se muestran todos los resultados obtenidos de esta síntesis a través de la herramienta Design Vision de Synopsys, en la que se muestran los tipos de datos obtenidos en función del número de bits del diseño:

	8 bits	16 bits	32 bits	64 bits	128 bits
Ports	8	12	20	36	68
Nets	122	204	258	458	842
Cells	91	160	144	248	440
References	24	28	22	23	25
Comb. Area (µm²)	1.043	1.930	3.701	7.184	13.981
No comb. Area (µm²)	589	887	1.485	2.679	5.068
Net area (µm²)	56	104	159	354	913
Total cell area (µm²)	1.632	2.817	5.186	9.863	19.048
Total area (µm²)	1.688	2.921	5.345	10.216	19.961
Consumo total (mW)	0,11	0,21	0,39	1,06	1,45
Pérdidas (µW)	6,28	10,80	19,48	35,98	66,90

	196 bits	256 bits	512 bits	1024 bits
Ports	102	132	260	516
Nets	1331	1745	3461	6916
Cells	725	959	1907	3826
References	23	21	21	21
Comb. Area (µm²)	21.588	28.212	56.345	112.726
No comb. Area (µm²)	7.606	9.845	19.400	38.511
Net area (µm²)	1.153	1.624	3.743	8.836
Total cell area (µm²)	29.194	38.057	75.745	151.237
Total area (µm²)	30.347	39.681	79.488	160.073
Consumo total (mW)	1,47	1,59	3,15	6,41
Pérdidas (µW)	103,85	135,48	269,49	538,40

A continuación se muestra una gráfica con el número de puertas lógicas empleadas en cada diseño según los bits empleados a la entrada.

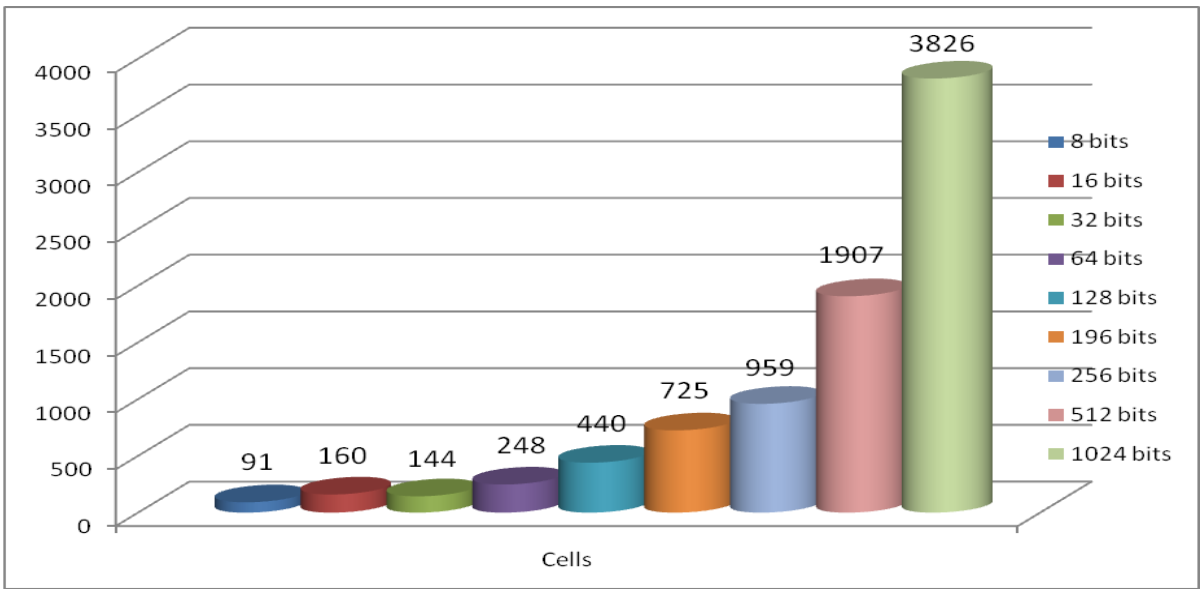

Figura 53.- Gráfica puertas lógicas algoritmo A.

En la siguiente gráfica se muestra el total del área según el número de bits empleados a la entrada con sus respectivas particiones en área correspondiente a la parte de la lógica combinacional y a la parte de la lógica no combinacional. Cabe destacar que para el total del área a parte de sumar estas dos partes correspondientes a la lógica, habría que añadirle la parte correspondiente al área del interconexionado.

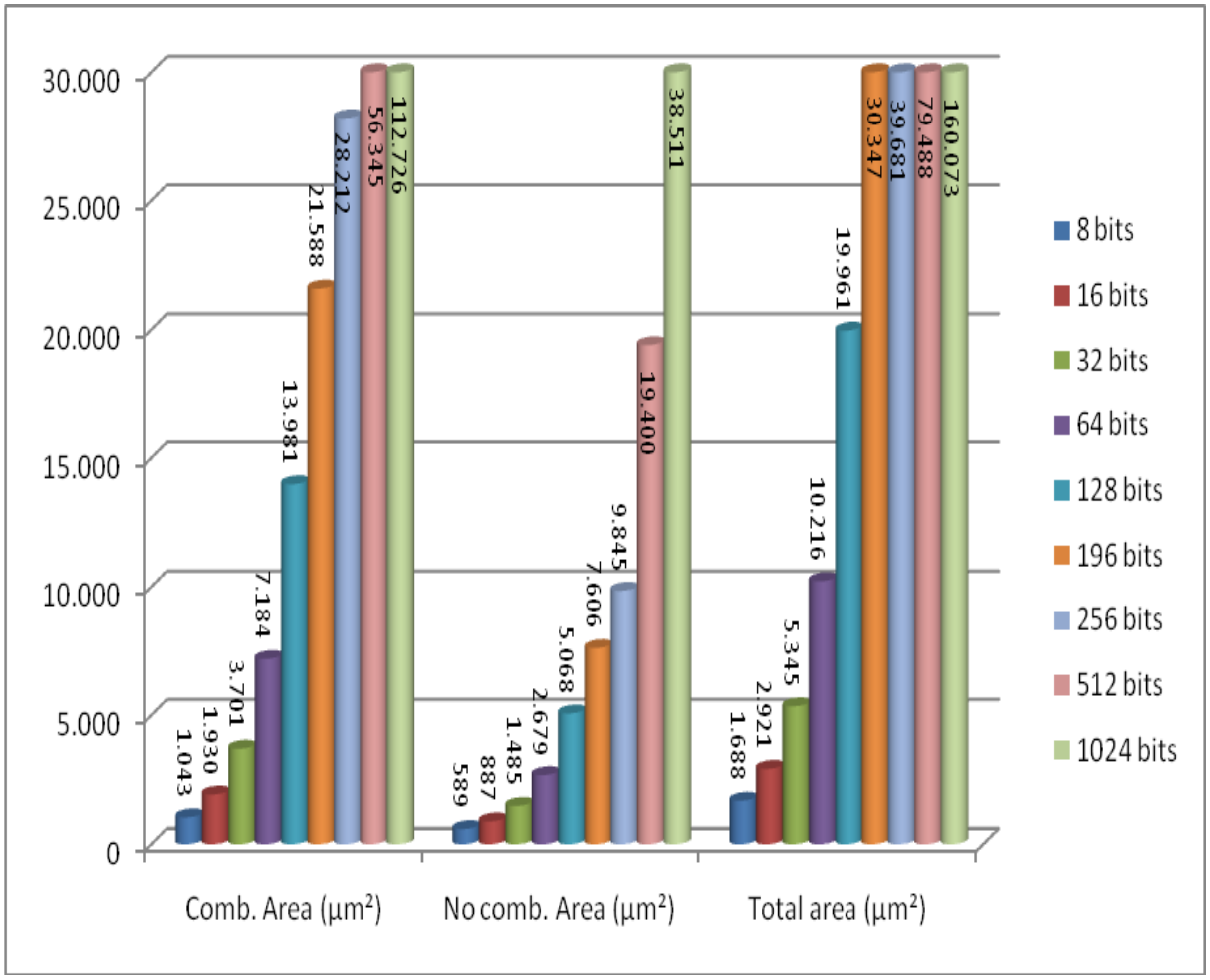

Figura 54.- Gráfica repartición área algoritmo A.

Se muestra la potencia que se consumiría (en unidades de miliwatios) por cada diseño en función del número de bits que se utilicen a la entrada por medio de la siguiente gráfica:

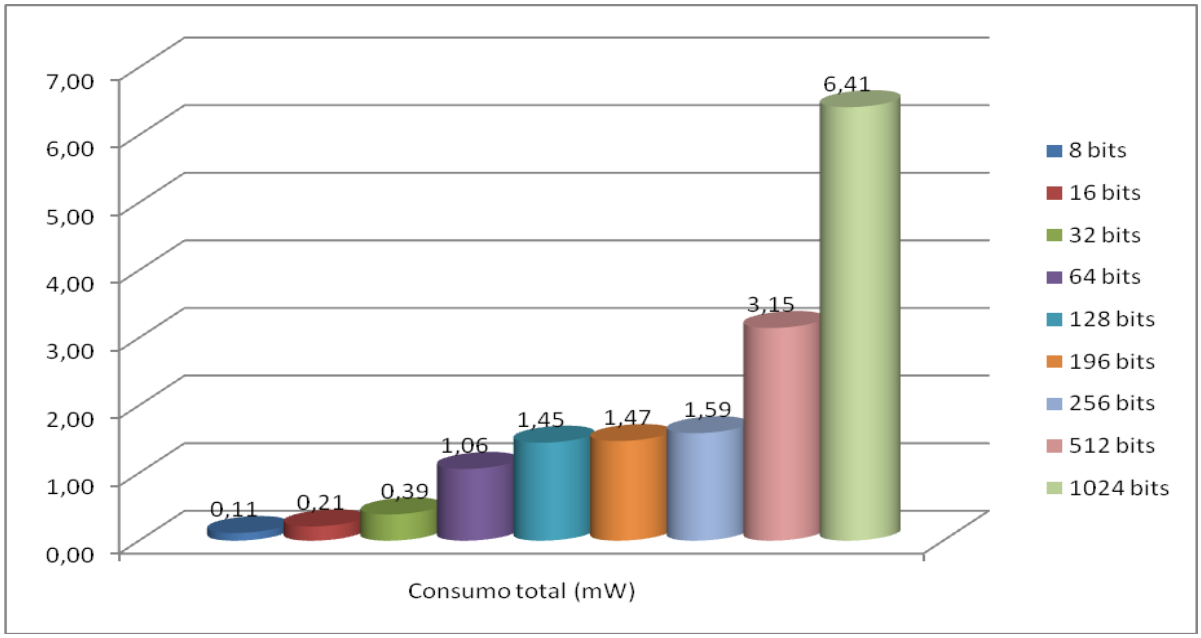

Figura 55.- Gráfica consumo algoritmo A.

4.2.6 Conclusiones

Una vez realizado el diseño de este generador ultraligero de números pseudo-aleatorios para este algoritmo A en el lenguaje de descripción hardware vhdl y sus posteriores implementaciones tanto para la tecnología de dispositivos programables FPGAs como para la tecnología de circuitos a medida ASICs, se pueden analizar los resultados obtenidos y llegar a una serie de conclusiones a partir de los mismos.

Destacar que mediante el ejecutable creado en Turbo C++, hemos comprobado que el diseño estaba realizado correctamente y posee un funcionamiento correcto de acuerdo a lo esperado.

De los resultados obtenidos en la implementación de dispositivos programables a partir del programa ISE, al igual que para el generador Blum Blum Shub se puede observar que comparando las dos tablas de los dos tipos de optimizaciones llevadas a cabo, en el caso de la optimización efectuada con un alto esfuerzo en área se consigue reducir ligeramente el tamaño del área medida en LUTs, así como el número de flip-flops y el número de slices respecto a la optimización con un alto esfuerzo en requerimientos de tiempo; sin embargo, con la optimización en tiempo se consigue una frecuencia máxima ligeramente

mayor con lo que el periodo mínimo sería menor y se conseguiría aumentar la velocidad.

Si para esta misma síntesis de dispositivos programables se compara ahora en función del número de bits a tratar en el proceso, se comprueba que también al igual que en el generador Blum Blum Shub a medida que aumenta el número de bits se va incrementando exponencialmente tanto el tamaño del área como el número de flip-flops y slices (figuras 51 y 52), a cambio, se observa que a medida que se aumenta el número de bits a tratar se reduce sustancialmente la frecuencia máxima haciendo los procesos cada vez más lentos. Hay un dato que se sale de este patrón, la frecuencia máxima del caso de 8 bits para la optimización en área es menor que la de los casos de 16 y 32 bits siendo así más lento el proceso que en estos dos casos (figura 51), esto puede ser debido a que al intentar optimizar precisamente en área lo máximo posible se reduzca la propia velocidad.

En cuanto a la síntesis realizada para circuitos integrados a medida también se extrae la conclusión de que a medida que se aumenta el número de bits a tratar se incrementan las puertas lógicas, el interconexionado, el tamaño del área de estas celdas, ya sea en la parte correspondiente a la lógica combinacional como a la parte correspondiente a la lógica no combinacional.

Así mismo, el estudio realizado del consumo de la potencia medida en miliwatios nos aporta que se consume bastante más a medida que aumenta el número de bits para estos generadores de números aleatorios (si bien para los casos de 128, 196 y 258 bits se mantiene casi constante), esto mismo ocurre con las pérdidas asociadas a esta potencia aunque en magnitudes mucho más inferiores ya que son del orden de microwatios.

DISEÑO PRNG´S

(PSEUDO-RANDOM NUMBER GENERATION)

OPCIÓN B

ALGORITMO – IMPLEMENTACIÓN VHDL – SIMULACIÓN – COMPROBACIÓN LENGUAJE C – SÍNTESIS ISE Y SYNOPSYS – RESULTADOS – CONCLUSIONES

4.3 Opción B

4.3.1 Algoritmo

Para este siguiente apartado se realiza el estudio del siguiente algoritmo de los tres generadores de números pseudo-aleatorios ultraligeros, a éste le denominaremos opción B, y al igual que el primero también está basado en el uso de una función triangular.

Se muestra el correspondiente código C del algoritmo B sobre el cual se realizaron las pruebas para observar la calidad de la salida.

```
Opción B

   x[0]= x[0] + ((x[0]*x[0])|5);
   x[1]= x[1] + ((x[1]*x[1])|13);

//Non-linerar function

r1=24 // se podría bajar hasta 16 rondas


/* A - Function */
z[0]=x[0]^x[1];
for(i=0;i<r1+){z[0]=(z[0]<<1)+((z[0]+(0x56AB0A))>>1);}


/* B - Function */
y[0]=x[1]^z[0];
for(i=0;i<r1++){y[0]=(y[0]>>1)+(y[0]<<1)+y[0]+(0x72A4FB);}

z[0]=z[0]^y[0];


//Filter output
// z[0] es la salida del generador
// me quedo con los bits menos significativos que son los que varían mas
rápidamente
// para el ejemplo de variables de 32 bits me quedo con los 16 menos
significativos


    z[0]=z[0]&0x0000ffff;
```

Para este algoritmo se va a realizar el estudio para los mismos casos que en la primera opción, y que en función del tamaño de bits de la entrada ó semilla quedarán establecidos los bits de salida de la siguiente manera:

Variables de 8 bits de entrada -> 4 bits de salida
Variables de 16 bits de entrada -> 8 bits de salida
Variables de 32 bits de entrada -> 16 bits de salida
Variables de 64 bits de entrada -> 48 bits de salida
Variables de 128 bits de entrada -> 64 bits de salida
Variables de 196 bits de entrada -> 98 bits de salida
Variables de 256 bits de entrada -> 128 bits de salida
Variables de 512 bits de entrada -> 256 bits de salida
Variables de 1024 bits de entrada -> 512 bits de salida

Este algoritmo parte con las dos primeras sentencias iguales a las del algoritmo A por lo que al introducir el valor semilla realizará el mismo tipo de operaciones que en A y se obtendrá los valores de las constantes x[0] y x[1].

El valor r1 es el que se establece a los contadores de las dos funciones A y B, que son los que controlan el número de veces que se iterará en el proceso repitiendo las mismas operaciones. Este valor lo hemos fijado a 24 para este estudio de la implementación que vamos a realizar para dar mayor seguridad, pero según los estudios previos realizados en la calidad de la salida de estos números generados este valor de r1 se podría bajar hasta 16 manteniendo una seguridad mínima para que estos números pseudo-aleatorios sean criptográficamente seguros y se puedan así utilizar para el fin deseado.

La función A consiste en asignar inicialmente a z[0] el valor de hacer un XOR a nivel de bit[5] entre las constantes x[0] y x[1], para posteriormente iterar un número de veces igual al valor de r1 realizando una serie de operaciones para la señal z. Éstas serían un registro desplazamiento a la izquierda de z y un registro desplazamiento a la derecha de la suma de z más el número hexadecimal 56AB0A, una vez realizadas estas operaciones por separado se suman y se repite el proceso r1 veces variando cada vez el valor de la señal z al estar realimentada asignándole el valor de la suma de las operaciones por separado.

[5] Ver anexo B

La función B consiste en establecer inicialmente a y[0] el valor de la realización de un XOR a nivel de bit entre la constante x[1] y el valor obtenido de la señal z en el proceso de la función A. En este caso las operaciones que se realizarían por separado a esta señal y[0] serían un registro desplazamiento de y a la izquierda y otro a la derecha para posteriormente sumarlas entre sí junto al valor de la señal y[0] además del número hexadecimal 72A4FB. El valor de esta suma realimentaría al valor de la señal y[0] repitiéndose el proceso r1 veces.

A continuación a la señal z se le asigna el valor de realizar un XOR a nivel de bit entre el propio valor de z obtenido en la función A y el valor de la señal y[0] obtenido de la función B.

En último lugar se le aplica el filtro de salida a la señal z para quedarse con el número de bits deseados a la salida en función de los bits de la entrada según el caso de los estudiados que se esté tratando en ese momento, así se obtendría el valor de z[0] quedándose con los bits menos significativos (los bits de la derecha) que son los que varían más rápidamente y son los que interesan de cara a que sean lo más aleatorios posibles para una segura encriptación.

NOTA: Una sencilla forma de obtener distintos generadores de números aleatorios a los estudiados en este proyecto sería la de cambiar las constantes que aparecen en forma de números hexadecimales de 24 bits (56AB0A y 72A4FB para este algoritmo B y 56AB0A para el siguiente algoritmo C) por otras constantes en las que se cogiesen los valores decimales del número pi o también se podrían coger diferentes números aleatorios generados del generador de la página web www.random.org; aunque claro está se debería de pasar estos números enteros en el sistema de numeración decimal al correspondiente sistema de numeración en que se diseñe el código vhdl.

4.3.2 Implementación en código Vhdl

La implementación de este algoritmo B en vhdl se ha llevado a cabo mediante la creación de una máquina de estados. Dicha implementación se rige por la siguiente entidad:

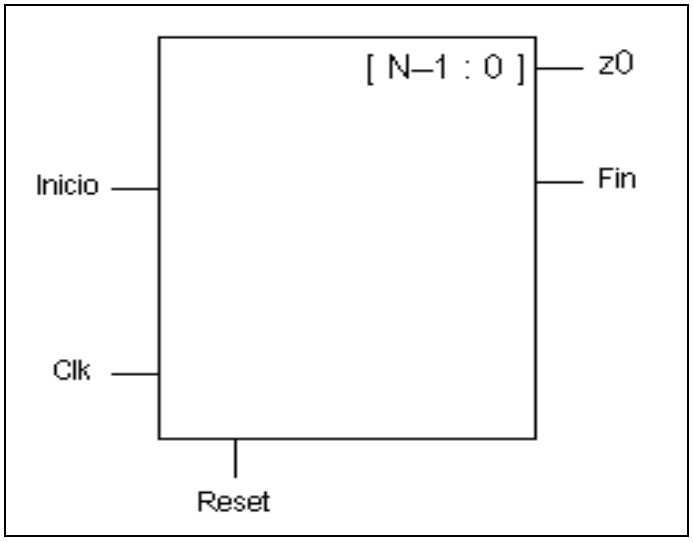

Figura 56.- Entidad algoritmo B.

De esta entidad al igual que en el algoritmo A, se observa que no hay señales de entrada específicas para la introducción de los datos o variables de entrada (bits que vamos a emplear para generar los números aleatorios), por lo que también emplearemos las semillas como constantes que no sean sintetizadas en el diseño. Estas constantes que son las que serán introducidas como valores semilla seed_0 y seed_1 se regirán por los mismos valores empleados en el algoritmo A, y como las dos primeras sentencias son comunes a los tres algoritmos, las constantes x0 y x1 que se calculan a partir de estas semillas también serán iguales que en el algoritmo A.

La máquina de estados empleada para la implementación de este algoritmo en vhdl sería la siguiente:

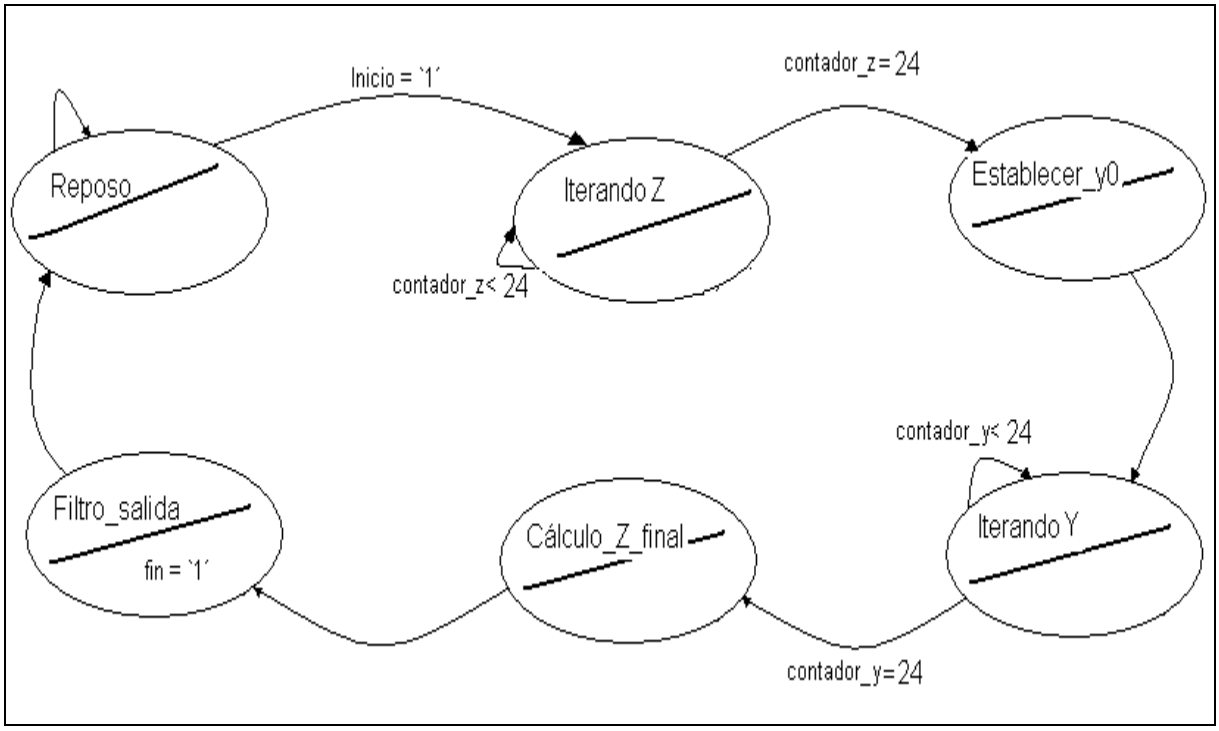

Figura 57.- Diagrama de estados algoritmo B.

a) REPOSO: Está a la espera del bit de Inicio para comenzar el proceso.
   Si es 1 go to Iterando, sino mantiene estado
b) ITERANDO Z: Realiza el mismo proceso durante 24 veces.
   Incrementa contador_z
   Si contador_z = 24 go to Establecer_y0, sino mantiene estado
c) ESTABLECER_y0: Calcula el valor inicial de la señal y0
   Go to Iterando Y
d) ITERANDO Y: Realiza el mismo proceso durante 24 veces.
   Incrementa contador_y
   Si contador_y = 24 go to Cálculo_Z_final, sino mantiene estado
e) CÁLCULO_Z_FINAL: Calcula el valor final de la señal z
   Go to Filtro_salida
f) FILTRO_SALIDA: Se queda con los bits menos significativos.
   Activa la salida fin

Go to Reposo

En esta implementación no es posible realizar los registros de desplazamiento de manera manual como en el código del algoritmo A[6], sino que es necesario usar los comandos shift_left y shift_right se han debido de cambiar las bibliotecas (`std_logic_unsigned y std_logic_arith`) por la biblioteca (`numeric_std`). Hay que tener en cuenta que a la hora de realizar el diseño vhdl con esta nueva biblioteca para definir las señales hay que usar unsigned en lugar de std_logic_vector y que no se puede usar la función conv_std_logic_vector sino que se debe introducir el número directamente como hexadecimal (x``……..´´).

Para esta implementación también hay que tener en cuenta otra consideración en la realización del diseño, y es que en esta opción B surgen problemas a la hora de simular para las variables 8 y 16 bits de entrada puesto que en las operaciones a realizar por el algoritmo aparecen los números hexadecimales (x"56AB0A" y x"72A4FB") que tienen un tamaño mayor a las variables a tratar (24 bits). Para solucionar esto se ha usado un *estado auxiliar* para truncar una señal auxiliar ajustándola al tamaño de la variable de entrada y una *señal auxiliar* de tamaño 24 bits para poder operar con cada uno de estos números respectivamente.

Para este problema se tiene la siguiente máquina de estados:

[6] Ver código en el apartado 2 del anexo A

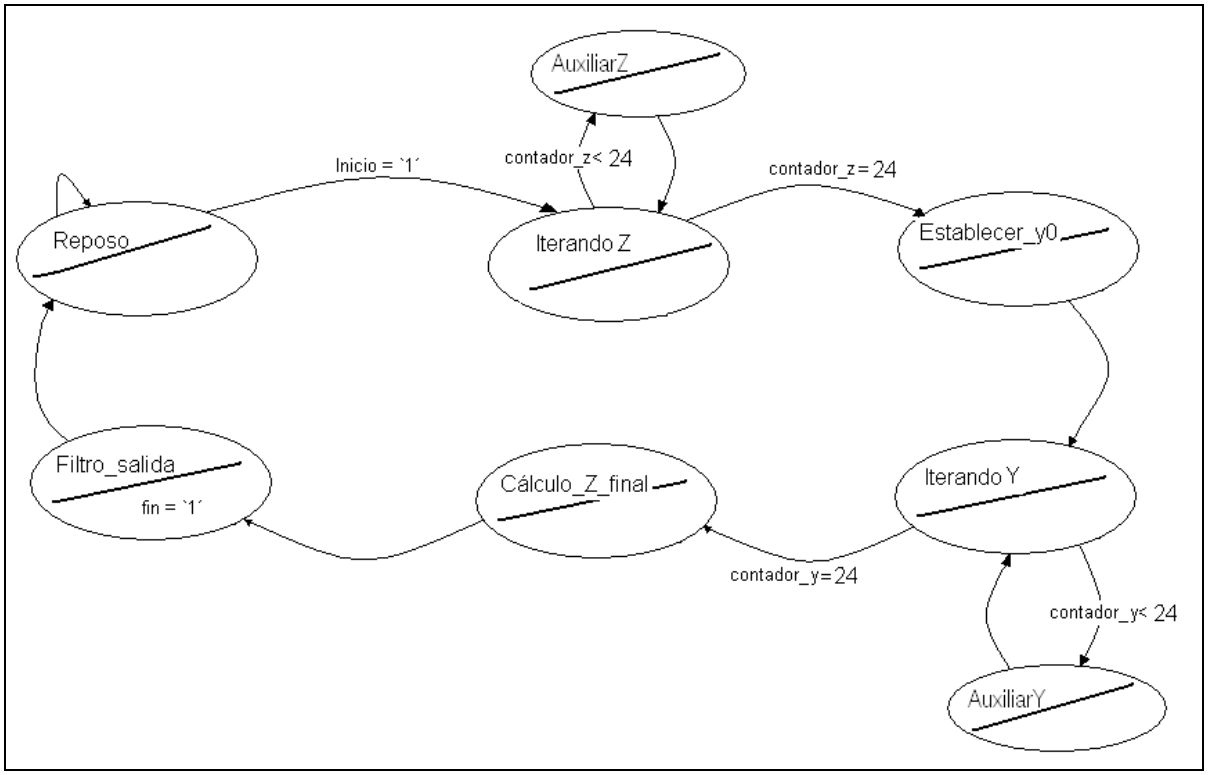

Figura 58.- Diagrama de estados para generadores de 8 y 16 bits de entrada del algoritmo B.

a) REPOSO: Está a la espera del bit de Inicio para comenzar el proceso.

   Si es 1 go to Iterando, sino mantiene estado

b) ITERANDO Z: Realiza el mismo proceso durante 24 veces.

   Incrementa contador_z

   Si contador_z = 24 go to Establecer_y0, sino go to Auxiliar Z

c) AUXILIAR Z: Opera el número hexadecimal x"56AB0A" con la señal auxiliar

   Go to Iterando Z

d) ESTABLECER_y0: Calcula el valor inicial de la señal y0

   Go to Iterando Y

e) ITERANDO Y: Realiza el mismo proceso durante 24 veces.

   Incrementa contador_y

   Si contador_y = 24 go to Cálculo_Z_final, sino go to Auxiliar Y

f) AUXILIAR Y: Opera el número hexadecimal x"72A4FB " con la señal auxiliar

   Go to Iterando Y

g) CÁLCULO_Z_FINAL: Calcula el valor final de la señal z

   Go to Filtro_salida

h) FILTRO_SALIDA: Se queda con los bits menos significativos.

   Activa la salida fin

   Go to Reposo

El código empleado en la implementación de este algoritmo B se muestra en el apartado 3 del anexo A, en él se muestra tanto el ejemplo para 32 bits de entrada como el ejemplo de 16 bits de entrada con los estados auxiliares. Para los demás casos mayores de 32 bits de entrada valdría con cambiar el valor del genérico N (que está puesto de tal modo que sea equivalente al nº de bits de salida) en el ejemplo de 32 bits, y para el caso de 8 bits de entrada habría que cambiar este genérico N en el ejemplo de 16 bits.

De la arquitectura empleada en la implementación de este algoritmo B se obtiene el siguiente esquema de componentes:

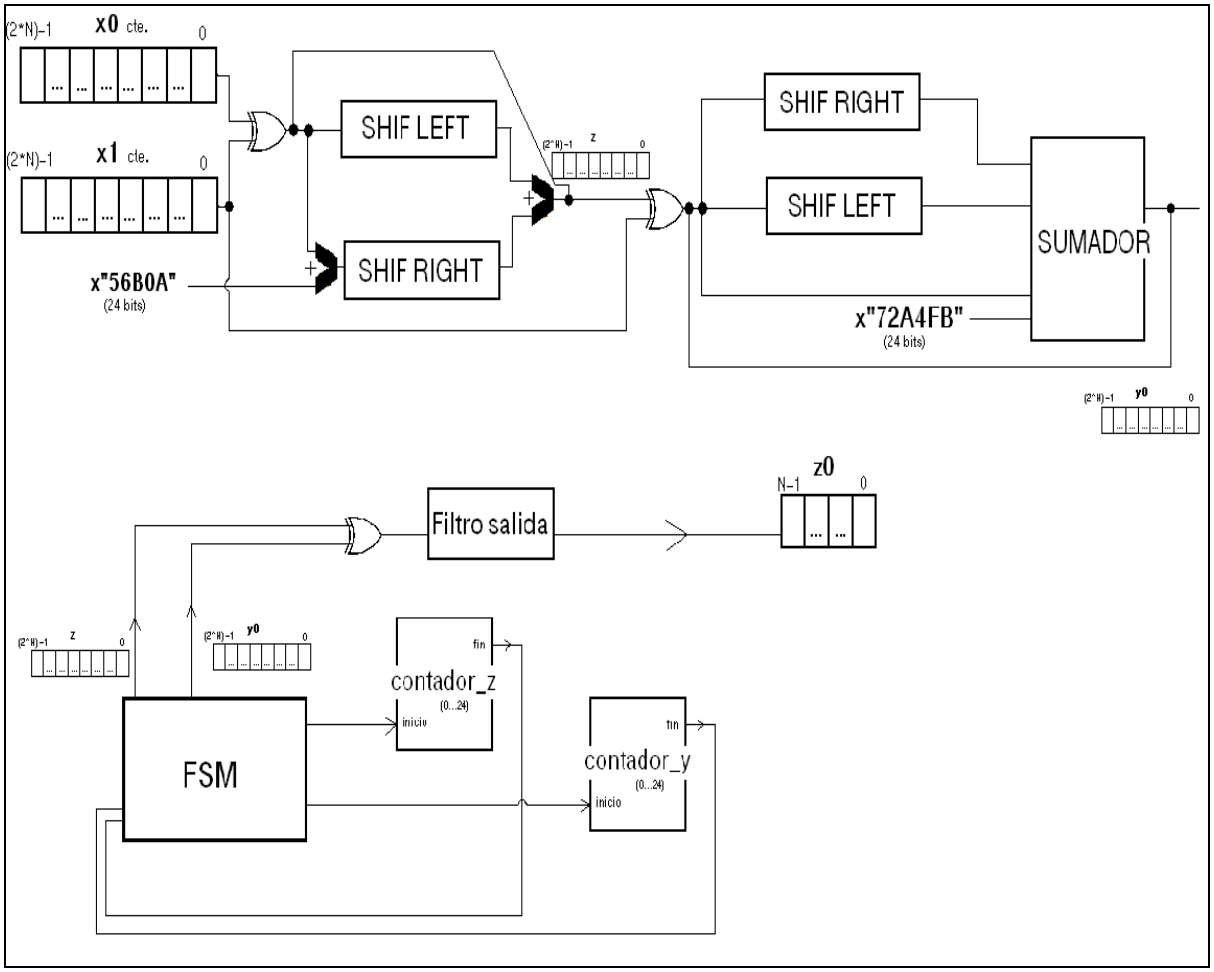

Figura 59.- Esquema de componentes algoritmo B.

De este esquema se observa que hay un control (FSM) que gestiona el proceso mediante el uso de dos contadores que van de 0 a 24, y que una vez realizado todo el proceso, el control envía como salida las señales z e y0 que son operadas por un XOR para posteriormente filtrarse y obtener la salida z0.

El proceso gestionado por la FSM contiene un sumador de cuatro entradas, otros dos sumadores de dos entradas, dos XOR, dos registros de desplazamiento a la izquierda, dos registros de desplazamiento a la derecha, también se observa el empleo de las constantes x0 y x1 y de los números hexadecimales utilizados en este algoritmo B.

4.3.3 Simulación

En este apartado se procede a la simulación de este algoritmo B, donde en primer lugar se observará a modo de ejemplo la simulación para 32 bits de entrada que genera un número aleatorio en z0.

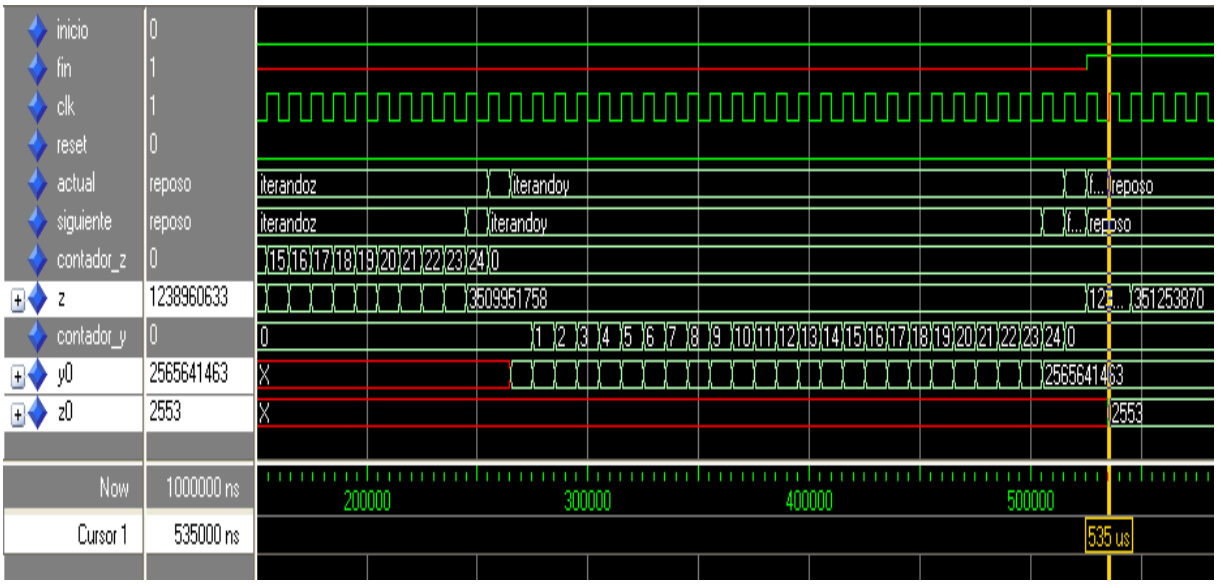

Figura 60.- Simulación 1 algoritmo B.

En la siguiente simulación se observa como el bit de inicio activa el comienzo del algoritmo, como se realiza la función A a través del estado Iterando Z y como el contador_z se va incrementando hasta su valor máximo 24 en el que cambia al estado de Establecer_y0.

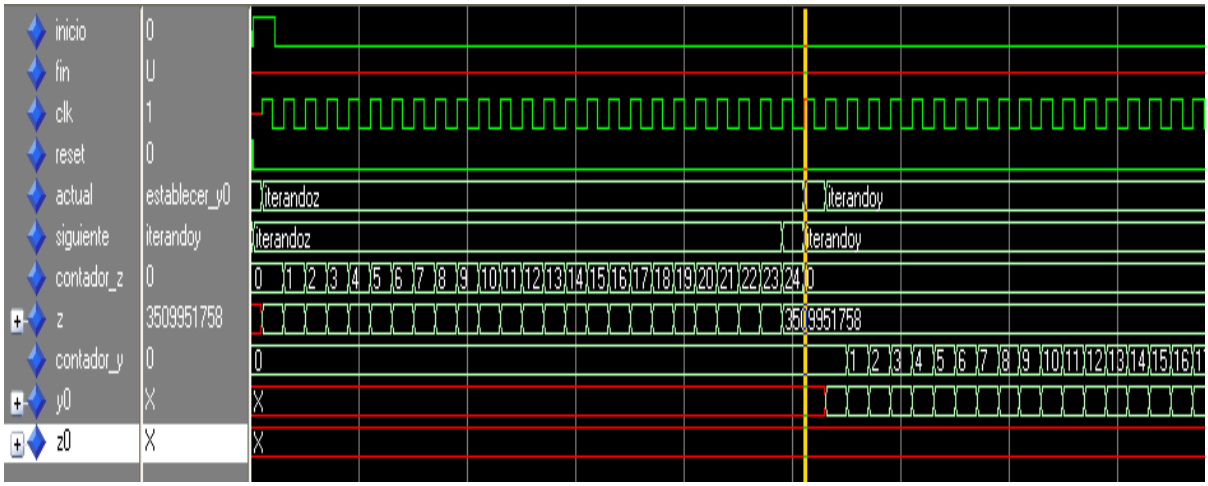

Figura 61.- Simulación 2 algoritmo B.

A continuación se observa como se establece el valor inicial de la señal y0 y como la señal z permanece fija. Se realiza la función B a través del estado Iterando Y en el que se van generando los números de la señal y0.

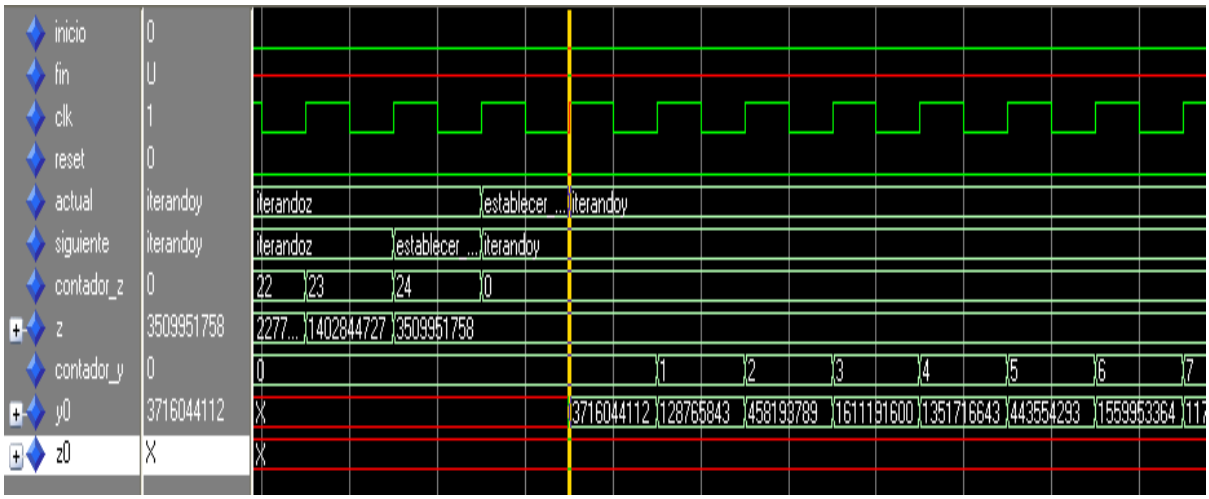

Figura 62.- Simulación 3 algoritmo B.

En la siguiente simulación se observa como se realiza el cálculo final de la señal z, valor al cual se le realiza el filtro de salida.

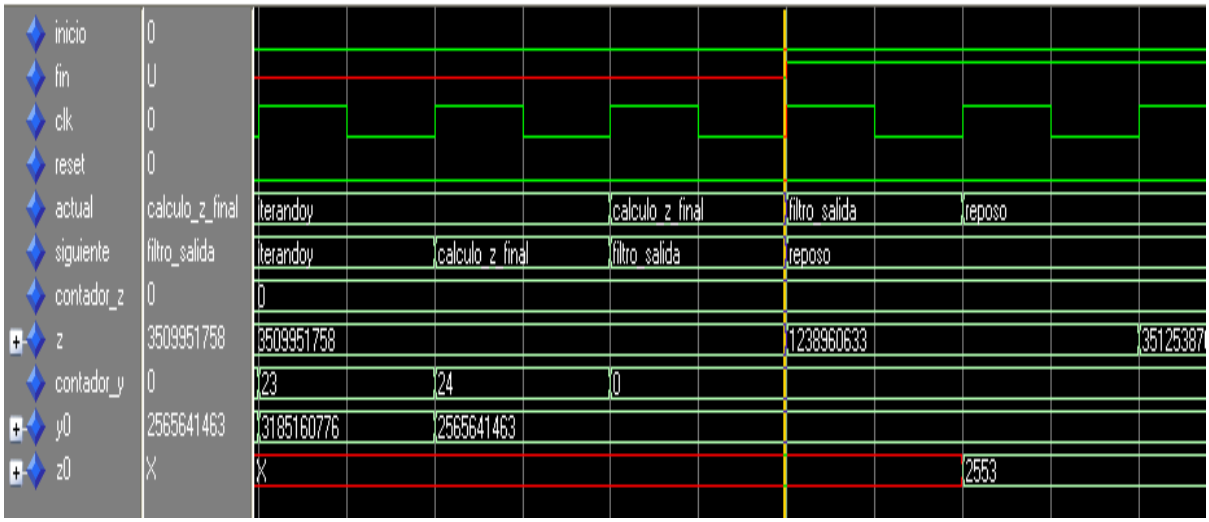

Figura 63.- Simulación 4 algoritmo B.

Vemos esta misma simulación pero en binario para observar mejor como actúa el XOR en el cálculo final de z y el filtro de salida quedándose con los 16 bits menos significativos que son los de la derecha y los que varían más rápidamente para generar los números pseudo-aleatorios.

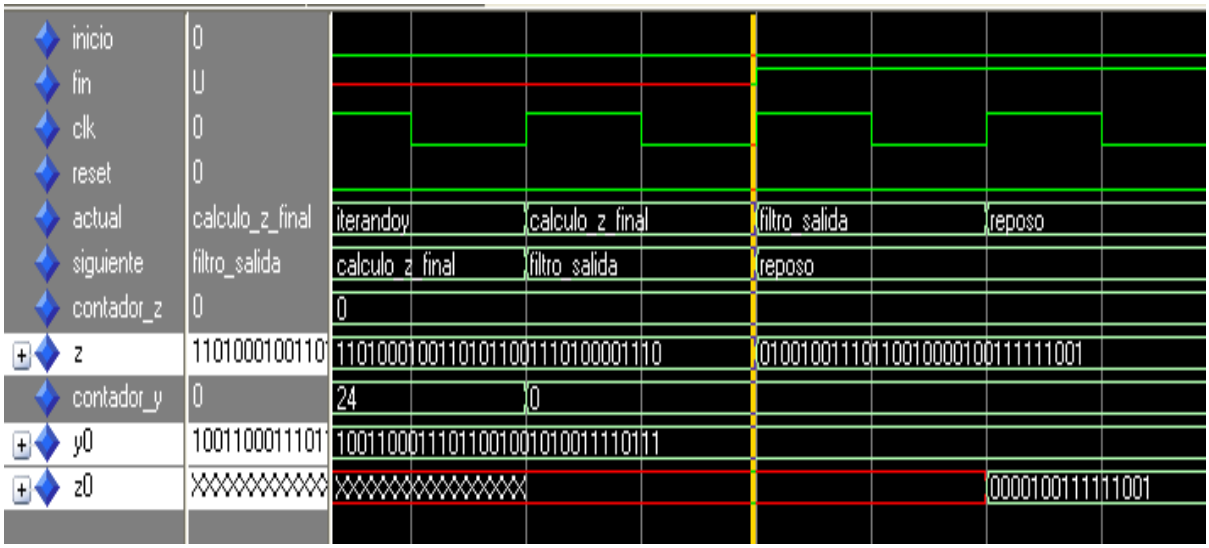

Figura 64.- Simulación 5 algoritmo B.

Ahora se simula exhaustivamente uno de los casos problemáticos como es el caso de 16 bits de entrada para el que es necesario el empleo de los estados auxiliares para poder operar con los números hexadecimales que son mayores que las señales a tratar, puesto que estas señales z e y0 son de 16 bits mientras que estos números hexadecimales son de 24 bits. En primer lugar se tiene la

simulación completa en la que se observa lo que tarda en realizarse el algoritmo completo y el número generado para las semillas introducidas como ejemplo.

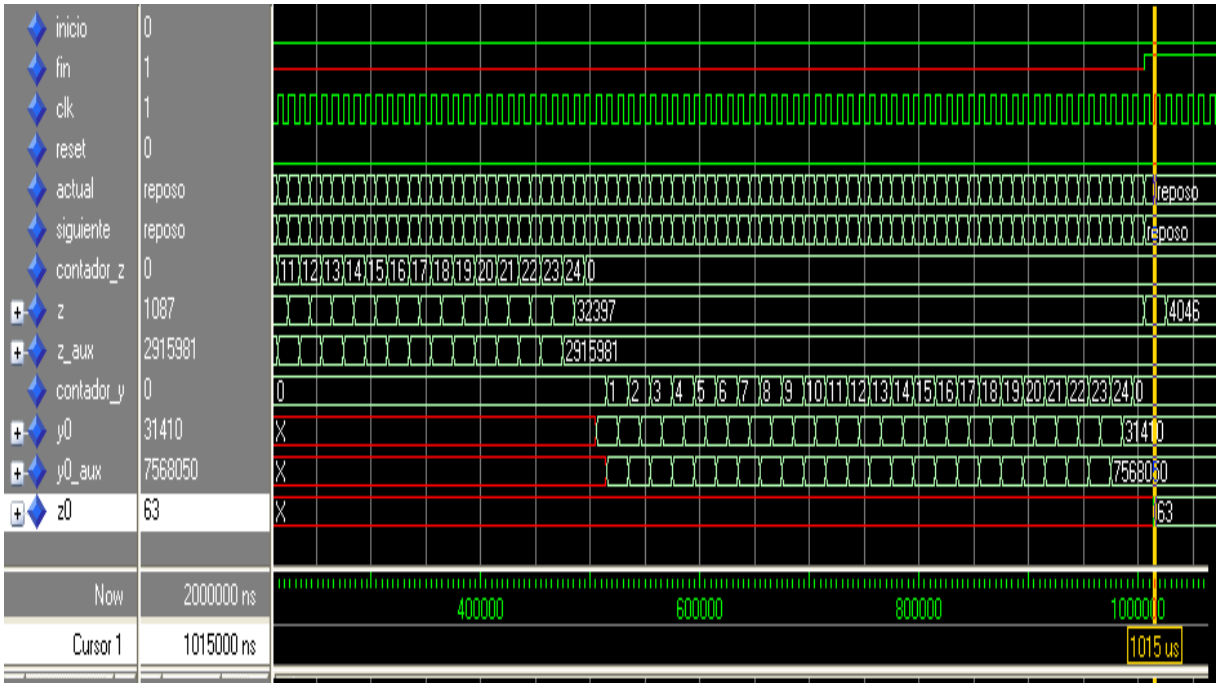

Figura 65.- Simulación 6 algoritmo B.

Se observa como funciona el auxiliar z en el que va cambiando de estado continuamente pasando de Auxiliar Z en el que opera con el número hexadecimal en la señal z_aux, a Iterando Z en el que se va truncando ese valor obtenido ajustándolo al tamaño de bits de la señal z.

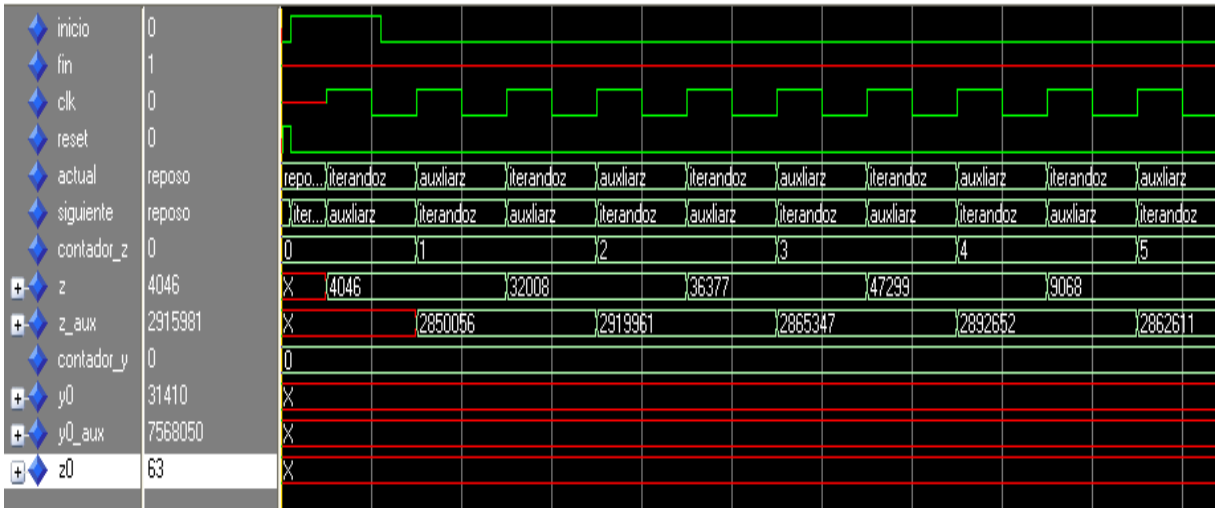

Figura 66.- Simulación 7 algoritmo B.

Del mismo modo se observa hasta cuando el contador_z llega a 24 cambia de estado para establecer y0 y como se realiza la función B, mediante el nuevo auxiliar y0 y la señal auxiliar y0_aux, de manera análoga al auxiliar z.

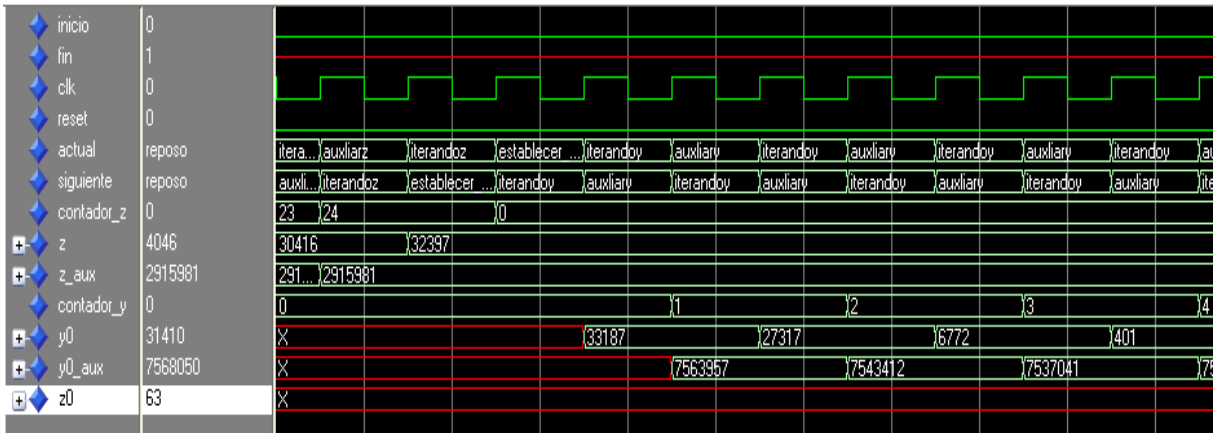

Figura 67.- Simulación 8 algoritmo B.

Se simula con las señales en binario para observar el mayor tamaño de las señales auxiliares, como se efectúa el XOR entre las señales z e y0 en el estado Cálculo final y el filtro de salida a este valor de z obtenido.

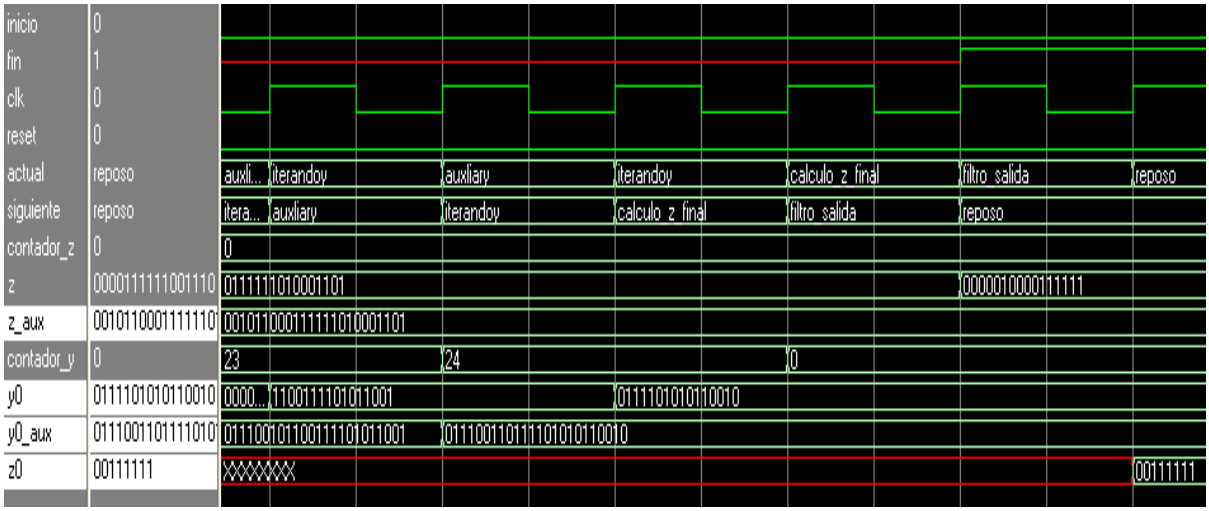

Figura 68.- Simulación 9 algoritmo B.

La simulación para el caso de 8 bits de entrada sería como la observada para el caso de 16 bits, se simula para ver el tiempo de ejecución y el número generado a la salida z0.

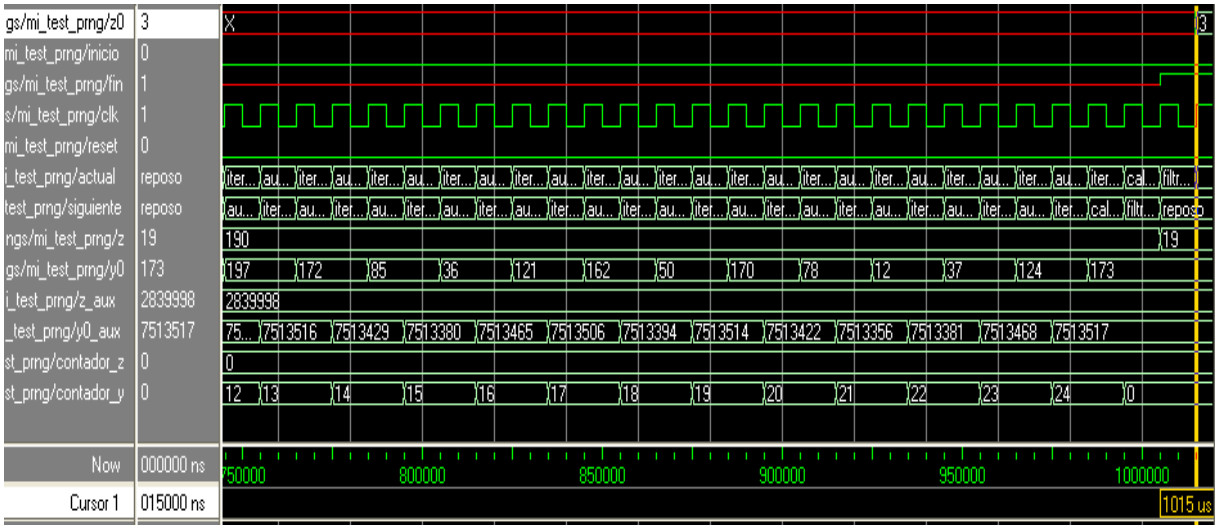

Figura 69.- Simulación 10 algoritmo B.

En estas simulaciones posteriores vamos a observar los resultados y el tiempo de ejecución de los casos mayores de 32 bits de entrada que tienen un proceso análogo a éste.

Primero el caso de 64 bits de entrada que genera un número pseudo-aleatorio de 48 bits en la salida z0.

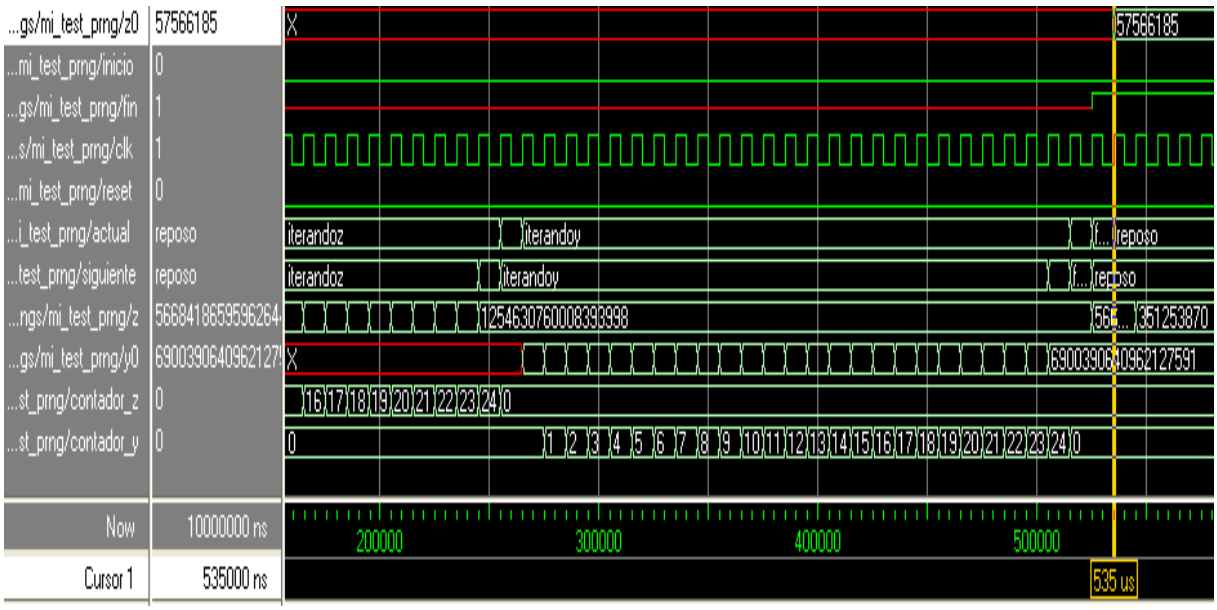

Figura 70.- Simulación 11 algoritmo B.

Simulación para 128 bits de entrada => 64 bits de salida z0. Según vamos aumentando el número de bits hay que tener en cuenta una consideración muy importante, y es que al usar una biblioteca distinta (`numeric_std`) en el diseño vhdl y no poder utilizar la función conv_std_logic_vector, al definir las constantes como números decimales hay que añadir ceros a la izquierda del número hexadecimal para adaptarlo al tamaño de cada caso de simulación que se realice según su número de bits a la entrada.

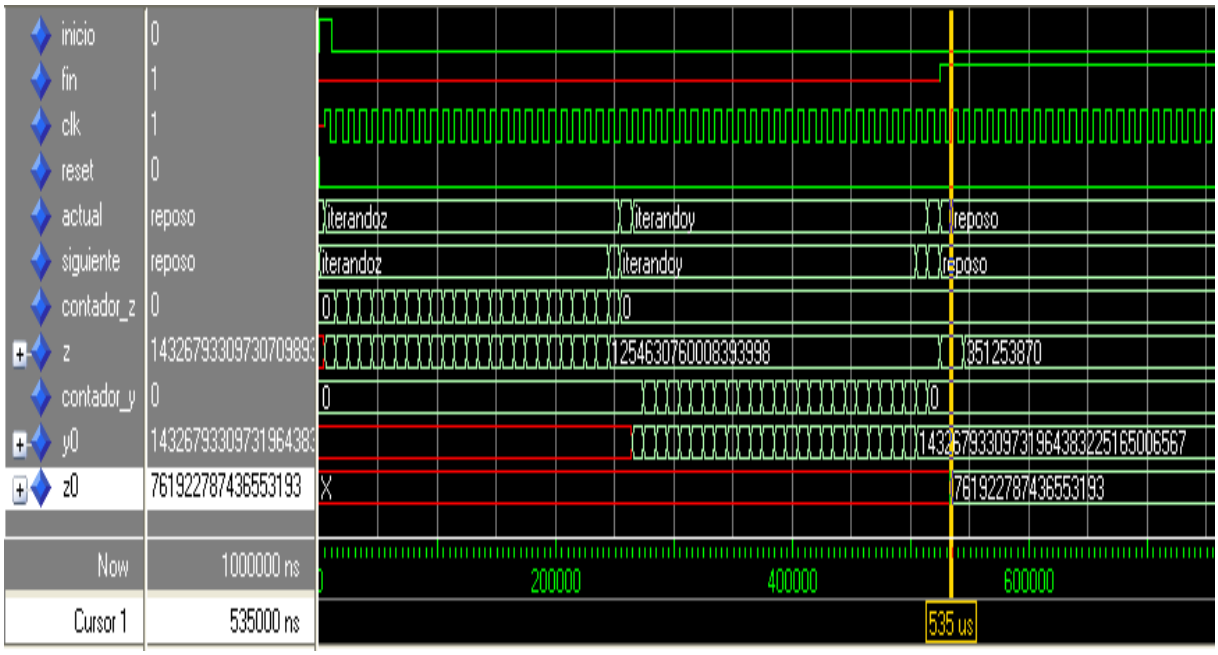

Figura 71.- Simulación 12 algoritmo B.

Simulación para 196 bits de entrada => 98 bits de salida z0.

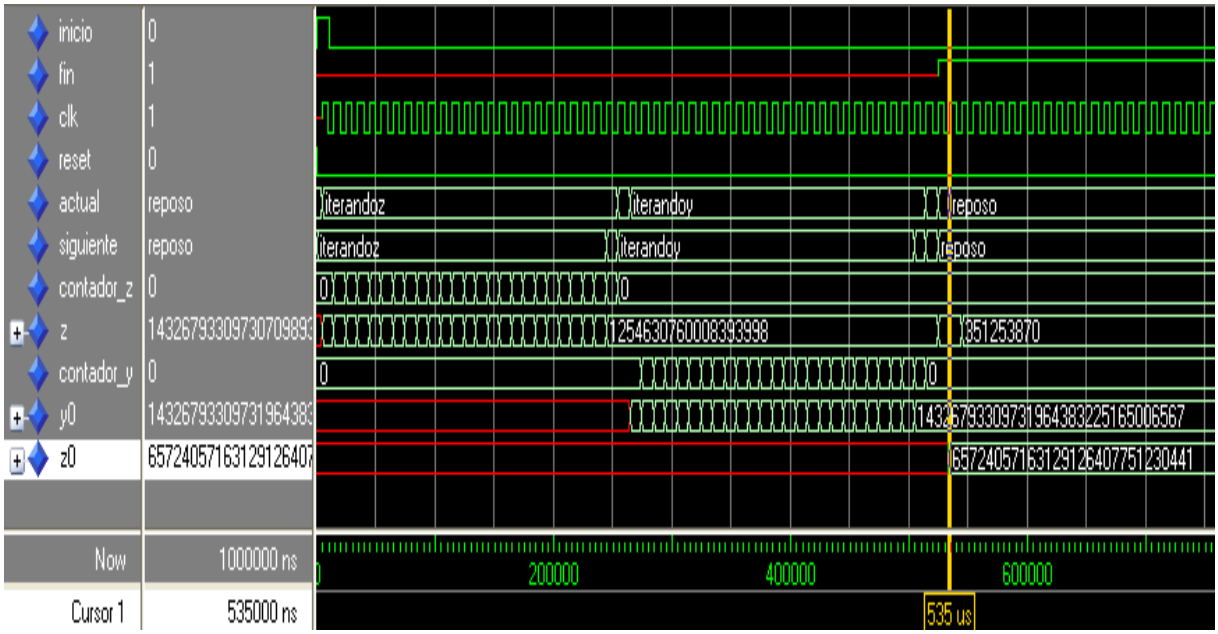

Figura 72.- Simulación 13 algoritmo B.

Simulación para 256 bits de entrada => N = 128 bits de salida z0.

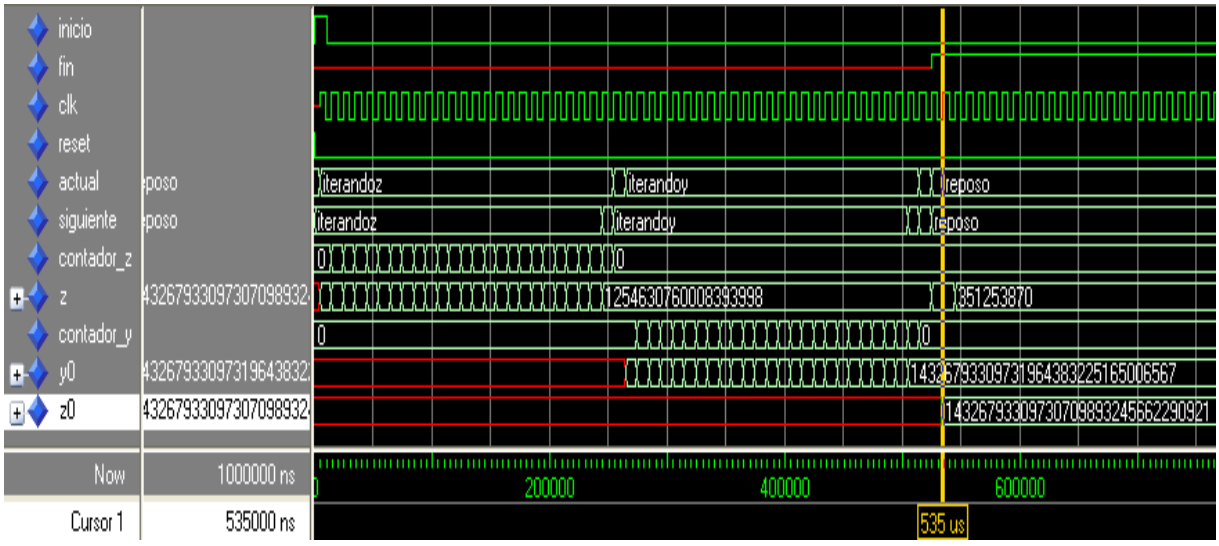

Figura 73.- Simulación 14 algoritmo B.

Simulación para 512 bits de entrada => N = 256 bits de salida z0.

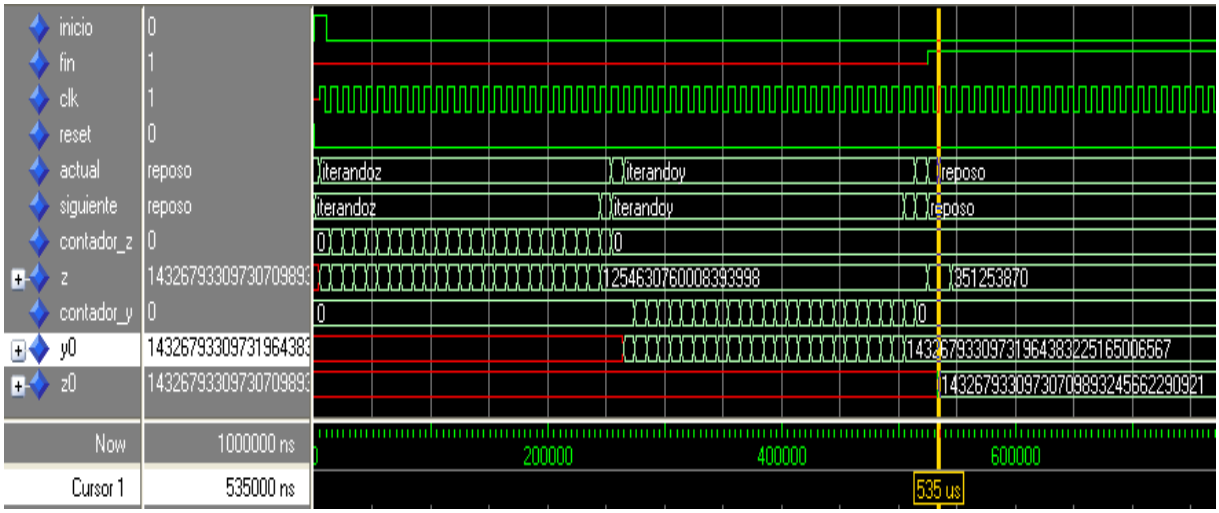

Figura 74.- Simulación 15 algoritmo B.

Simulación de 1024 bits de entrada => N = 512 bits de salida z0.

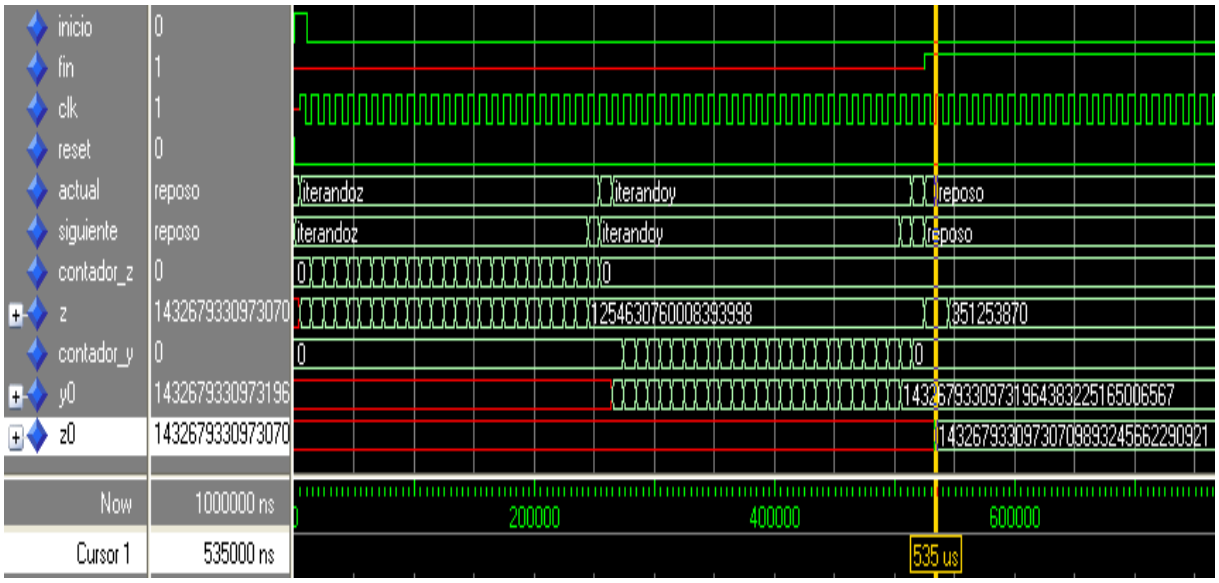

Figura 75.- Simulación 16 algoritmo B.

De estas simulaciones se comprueba que para los casos de 32 bits y mayores se emplea el mismo tiempo de ejecución del algoritmo que es de 535 µs. Para los casos problemáticos de 8 y 16 bits de entrada se emplean 1015 µs un tiempo mayor como es lógico, ya que emplea un mayor número de estados y operaciones.

A su vez de estas simulaciones se extraen los siguientes resultados como números generados en la salida z0.

Nº de bits de entrada	Genérico N de bits	Nº de bits de la salida z0	Nº generado en la salida z0
8	4	4	3
16	8	8	63
32	16	16	2.553
64	32	48	57.566.185
128	64	64	761.922.787.436.553.193
196	98	98	65.724.057.163.129.126.407.751.230.441
256	128	128	14.326.793.309.730.709.893.245.662.290.921
512	256	256	14.326.793.309.730.709.893.245.662.290.921
1024	512	512	14.326.793.309.730.709.893.245.662.290.921

Se observa que el resultado se repite para los casos de un número de bits muy elevado a la entrada, puesto que para las constantes no se permiten introducir números tan elevados en vhdl; el número entero más elevado soportado por vhdl es el 2.147.483.647.

4.3.4 Comprobación lenguaje C

Al igual que para el algoritmo A se ha procedido a comprobar los resultados obtenidos de estas simulaciones de Modelsim mediante Turbo C++.

Se muestra el código para el cual se obtiene el resultado en 16 bits de salida al introducir cualquier número que contenga como máximo 32 bits de entrada.

```
#include <stdio.h>

/* Opcion B. Variables de 32 bits de entrada y 16 bits de salida. */

main()

{

 long unsigned x0, x1, y0, z0;

 int varcon, i, r1;
```

```
do {
        printf("\nIntroducir un valor entero para x0: ");
        scanf("%lu", &x0);
        printf("\nIntroducir un valor entero para x1: ");
        scanf("%lu", &x1);
        printf("\nValor inicial de x0: %lu", x0);
        printf("\nValor inicial de x1: %lu", x1);
        x0= x0 + ((x0*x0) | 5);
        x1= x1 + ((x1*x1) | 13);
        printf("\nValor actual de x0: %lu", x0);
        printf("\nValor actual de x1: %lu", x1);
        //Non-linear function
        r1=24; /* Cambiando r1, se podria variar el numero de rondas */
        /* A - Function */
        z0=x0^x1;
        for (i=0; i<r1; i++) {
                z0=(z0<<1)+((z0+(0x56AB0A))>>1);
                }
        /* B - Function */
        y0=x1^z0;
        for(i=0; i<r1; i++) {
                y0=(y0>>1)+(y0<<1)+y0+(0x72A4FB);
                }
        z0=z0^y0;
        //Filter output
        //z0 es la salida del generador,
        //nos quedamos con los bits menos significativos que son los que varian
        //mas rapidamente.
        //para el ejemplo de variables de 16 bits, nos quedamos con los 8 bits
```

```
        //menos significativos.
        printf("\nValor decimal de z0: %lu", z0);
        printf("\nValor hexadecimal de z0: %lx", z0);
        z0=z0&0x0000ffff;
        printf("\nValor hexadecimal despues de la mascara de z0: %lx", z0);
        printf("\nValor decimal despues de la mascara de z0: %lu", z0);
        printf("\nPara continuar introducir 1, para parar introducir 0: ");
        scanf("%d", &varcon);
        } while (varcon != 0);
 printf("\nFin del programa");
 }
```

También se muestra el código en el cual se introducen 16 bits de entrada y se obtienen 8 bits de salida.

```
#include <stdio.h>
/* Opcion B. Variables de 16 bits de entrada y 8 bits de salida. */
main()
{
 unsigned x0, x1, y0, z0, varcon, i, r1;
 do {
        printf("\nIntroducir un valor entero para x0: ");
        scanf("%u", &x0);
        printf("\nIntroducir un valor entero para x1: ");
        scanf("%u", &x1);
        printf("\nValor inicial de x0: %u", x0);
        printf("\nValor inicial de x1: %u", x1);
        x0= x0 + ((x0*x0) | 5);
        x1= x1 + ((x1*x1) | 13);
```

```
printf("\nValor actual de x0: %u", x0);
printf("\nValor actual de x1: %u", x1);
//Non-linear function
r1=24; /* Cambiando r1, se podria variar el numero de rondas */
/* A - Function */
z0=x0^x1;
for (i=0; i<r1; i++) {
        z0=(z0<<1)+((z0+(0x56AB0A))>>1);
        }
/* B - Function */
y0=x1^z0;
for(i=0; i<r1; i++) {
        y0=(y0>>1)+(y0<<1)+y0+(0x72A4FB);
        }
z0=z0^y0;
//Filter output
//z0 es la salida del generador,
//nos quedamos con los bits menos significativos que son los que varian
//mas rapidamente.
//para el ejemplo de variables de 16 bits, nos quedamos con los 8 bits
//menos significativos.
printf("\nValor decimal de z0: %u", z0);
printf("\nValor hexadecimal de z0: %x", z0);
z0=z0&0x00ff;
printf("\nValor hexadecimal despues de la mascara de z0: %x", z0);
printf("\nValor decimal despues de la mascara de z0: %u", z0);
printf("\nPara continuar introducir 1, para parar introducir 0: ");
scanf("%d", &varcon);
} while (varcon != 0);
```

```
printf("\nFin del programa");
}
```

La diferencia entre estos dos códigos radica en que, al igual que en el algoritmo A, para la opción de 32 bits de entrada hay que definir a los valores de x0, x1 y z0 como *long int* y no como entero simple como en la opción de 16 bits.

Se comprobó que el resultado de la simulación en Modelsim de la implementación de esta opción B de generador de números pseudo-aleatorios en vhdl concordaba con el resultado obtenido de la ejecución de este código tanto para 32 bits de entrada como para 16.

Al igual que en el algoritmo también se disponen de los ejecutables[7] para comprobar y obtener cuantos números se quieran con el generador de esta opción B. Para comprobar los resultados obtenidos de los ejecutables con el diseño en vhdl bastaría con cambiar los valores de las constantes seed_0 y seed_1 y calcular los de las constantes x0 y x1.

4.3.5 Síntesis

4.3.5.1 Síntesis ISE

Se muestra la síntesis realizada de este diseño del algoritmo B por medio del programa ISE.

Al igual que para el algoritmo A, el objetivo es implementar este diseño en una FPGA lo más optima posible en base a nuestros requerimientos, con lo que debe ser lo más pequeña posible.

[7] Estos ejecutables se encuentran en el cd adjunto a este proyecto

Para los casos de menor número de bits a la entrada se ha vuelto a escoger una FPGA de la familia Spartan 3, más concretamente el dispositivo XC3S50, consiguiendo con éste un gran avance para nuestra tecnología, al haber conseguido implementar nuestro diseño en una FPGA de las más pequeñas del mercado.

Property Name	Value
Product Category	General Purpose
Family	Spartan3
Device	XC3S50
Package	PQ208
Speed	-5

8 bits de entrada -> N = 4 bits de salida

Implementado con optimización en Área (Goal: Area, Effort: High)

Frecuencia máxima = 135,707 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	45	1,536	2%
Number of 4 input LUTs	125	1,536	8%
Logic Distribution			
Number of occupied Slices	67	768	8%
Number of Slices containing only related logic	67	67	100%
Number of Slices containing unrelated logic	0	67	0%
Total Number of 4 input LUTs	126	1,536	8%
Number used as logic	125		
Number used as a route-thru	1		
Number of bonded IOBs	8	124	6%
IOB Flip Flops	4		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal: Speed, Effort: High)

Frecuencia máxima = 177,099 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	49	1,536	3%
Number of 4 input LUTs	129	1,536	8%
Logic Distribution			
Number of occupied Slices	70	768	9%
Number of Slices containing only related logic	70	70	100%
Number of Slices containing unrelated logic	0	70	0%
Total Number of 4 input LUTs	129	1,536	8%
Number of bonded IOBs	8	124	6%
IOB Flip Flops	4		
Number of GCLKs	1	8	12%

16 bits de entrada -> N = 8 bits de salida

Implementado con optimización en Área

Frecuencia máxima = 135,543 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	77	1,536	5%
Number of 4 input LUTs	178	1,536	11%
Logic Distribution			
Number of occupied Slices	100	768	13%
Number of Slices containing only related logic	100	100	100%
Number of Slices containing unrelated logic	0	100	0%
Total Number of 4 input LUTs	193	1,536	12%
Number used as logic	178		
Number used as a route-thru	15		
Number of bonded IOBs	12	124	9%
IOB Flip Flops	8		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 136,412 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	77	1,536	5%
Number of 4 input LUTs	181	1,536	11%
Logic Distribution			
Number of occupied Slices	101	768	13%
Number of Slices containing only related logic	101	101	100%
Number of Slices containing unrelated logic	0	101	0%
Total Number of 4 input LUTs	196	1,536	12%
Number used as logic	181		
Number used as a route-thru	15		
Number of bonded IOBs	12	124	9%
IOB Flip Flops	8		
Number of GCLKs	1	8	12%

32 bits de entrada -> N = 16 bits de salida

Implementado en Área

Frecuencia máxima = 94,885 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	77	1,536	5%
Number of 4 input LUTs	308	1,536	20%
Logic Distribution			
Number of occupied Slices	178	768	23%
Number of Slices containing only related logic	178	178	100%
Number of Slices containing unrelated logic	0	178	0%
Total Number of 4 input LUTs	349	1,536	22%
Number used as logic	308		
Number used as a route-thru	41		
Number of bonded IOBs	20	124	16%
IOB Flip Flops	16		
Number of GCLKs	1	8	12%

Implementado en Velocidad

Frecuencia máxima = 106,055 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	77	1,536	5%
Number of 4 input LUTs	329	1,536	21%
Logic Distribution			
Number of occupied Slices	188	768	24%
Number of Slices containing only related logic	188	188	100%
Number of Slices containing unrelated logic	0	188	0%
Total Number of 4 input LUTs	370	1,536	24%
Number used as logic	329		
Number used as a route-thru	41		
Number of bonded IOBs	20	124	16%
IOB Flip Flops	16		
Number of GCLKs	1	8	12%

64 bits de entrada -> 48 bits de salida

Implementado con optimización en Área

Frecuencia máxima = 82,034 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	141	1,536	9%
Number of 4 input LUTs	566	1,536	36%
Logic Distribution			
Number of occupied Slices	338	768	44%
Number of Slices containing only related logic	338	338	100%
Number of Slices containing unrelated logic	0	338	0%
Total Number of 4 input LUTs	671	1,536	43%
Number used as logic	566		
Number used as a route-thru	105		
Number of bonded IOBs	36	124	29%
IOB Flip Flops	32		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 89,245 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	141	1,536	9%
Number of 4 input LUTs	564	1,536	36%
Logic Distribution			
Number of occupied Slices	339	768	44%
Number of Slices containing only related logic	339	339	100%
Number of Slices containing unrelated logic	0	339	0%
Total Number of 4 input LUTs	671	1,536	43%
Number used as logic	564		
Number used as a route-thru	107		
Number of bonded IOBs	36	124	29%
IOB Flip Flops	32		
Number of GCLKs	1	8	12%

128 bits de entrada -> N = 64 bits de salida

Implementado con optimización en Área

Frecuencia máxima = 63,524 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	269	1,536	17%
Number of 4 input LUTs	1,079	1,536	70%
Logic Distribution			
Number of occupied Slices	667	768	86%
Number of Slices containing only related logic	667	667	100%
Number of Slices containing unrelated logic	0	667	0%
Total Number of 4 input LUTs	1,312	1,536	85%
Number used as logic	1,079		
Number used as a route-thru	233		
Number of bonded IOBs	68	124	54%
IOB Flip Flops	64		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 67,764 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	269	1,536	17%
Number of 4 input LUTs	1,076	1,536	70%
Logic Distribution			
Number of occupied Slices	670	768	87%
Number of Slices containing only related logic	670	670	100%
Number of Slices containing unrelated logic	0	670	0%
Total Number of 4 input LUTs	1,311	1,536	85%
Number used as logic	1,076		
Number used as a route-thru	235		
Number of bonded IOBs	68	124	54%
IOB Flip Flops	64		
Number of GCLKs	1	8	12%

196 bits de entrada -> N = 98 bits de salida

Para estas implementaciones siguientes de 196 y 256 bits de entrada ha sido necesario la utilización de una FPGA de la familia Spartan 3 siendo el dispositivo XC3S200 el más pequeño en el que podíamos introducir nuestro diseño para cumplir con nuestros requerimientos de área.

Property Name	Value
Product Category	General Purpose
Family	Spartan3
Device	XC3S200
Package	PQ208
Speed	-5

Implementado con optimización en Área

Frecuencia máxima = 50,914 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	405	3,840	10%
Number of 4 input LUTs	1,621	3,840	42%
Logic Distribution			
Number of occupied Slices	1,035	1,920	53%
Number of Slices containing only related logic	1,035	1,035	100%
Number of Slices containing unrelated logic	0	1,035	0%
Total Number of 4 input LUTs	1,990	3,840	51%
Number used as logic	1,621		
Number used as a route-thru	369		
Number of bonded IOBs	102	141	72%
IOB Flip Flops	98		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 53,963 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	405	3,840	10%
Number of 4 input LUTs	1,619	3,840	42%
Logic Distribution			
Number of occupied Slices	1,033	1,920	53%
Number of Slices containing only related logic	1,033	1,033	100%
Number of Slices containing unrelated logic	0	1,033	0%
Total Number of 4 input LUTs	1,990	3,840	51%
Number used as logic	1,619		
Number used as a route-thru	371		
Number of bonded IOBs	102	141	72%
IOB Flip Flops	98		
Number of GCLKs	1	8	12%

256 bits de entrada -> N = 128 bits de salida

Implementado con optimización en Área

Frecuencia máxima = 43,533 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	526	3,840	13%
Number of 4 input LUTs	2,101	3,840	54%
Logic Distribution			
Number of occupied Slices	1,315	1,920	68%
Number of Slices containing only related logic	1,315	1,315	100%
Number of Slices containing unrelated logic	0	1,315	0%
Total Number of 4 input LUTs	2,590	3,840	67%
Number used as logic	2,101		
Number used as a route-thru	489		
Number of bonded IOBs	132	141	93%
IOB Flip Flops	128		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 45,743 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	525	3,840	13%
Number of 4 input LUTs	2,099	3,840	54%
Logic Distribution			
Number of occupied Slices	1,317	1,920	68%
Number of Slices containing only related logic	1,317	1,317	100%
Number of Slices containing unrelated logic	0	1,317	0%
Total Number of 4 input LUTs	2,590	3,840	67%
Number used as logic	2,099		
Number used as a route-thru	491		
Number of bonded IOBs	132	141	93%
IOB Flip Flops	128		
Number of GCLKs	1	8	12%

512 bits de entrada -> N = 256 bits de salida

Para esta implementación de 512 bits de entrada se ha utilizado el dispositivo XC3S2000 siendo el más pequeño en el que podíamos implementar nuestro diseño.

Property Name	Value
Product Category	General Purpose
Family	Spartan3
Device	XC3S2000
Package	FG456
Speed	-5

Implementado con optimización en Área

Frecuencia máxima = 26,988 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,039	40,960	2%
Number of 4 input LUTs	4,155	40,960	10%
Logic Distribution			
Number of occupied Slices	2,596	20,480	12%
Number of Slices containing only related logic	2,596	2,596	100%
Number of Slices containing unrelated logic	0	2,596	0%
Total Number of 4 input LUTs	5,156	40,960	12%
Number used as logic	4,155		
Number used as a route-thru	1,001		
Number of bonded IOBs	260	333	78%
IOB Flip Flops	256		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 27,725 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,039	40,960	2%
Number of 4 input LUTs	4,151	40,960	10%
Logic Distribution			
Number of occupied Slices	2,595	20,480	12%
Number of Slices containing only related logic	2,595	2,595	100%
Number of Slices containing unrelated logic	0	2,595	0%
Total Number of 4 input LUTs	5,154	40,960	12%
Number used as logic	4,151		
Number used as a route-thru	1,003		
Number of bonded IOBs	260	333	78%
IOB Flip Flops	256		
Number of GCLKs	1	8	12%

1024 bits de entrada -> N = 512 bits de salida

En esta implementación de 1024 bits de entrada se ha utilizado el dispositivo XC3S5000 siendo el más pequeño en el que podíamos implementar nuestro diseño.

Property Name	Value
Product Category	General Purpose
Family	Spartan3
Device	XC3S5000
Package	FG900
Speed	-5

Implementado con optimización en Área

Frecuencia máxima = 15,274 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	2,067	66,560	3%
Number of 4 input LUTs	8,258	66,560	12%
Logic Distribution			
Number of occupied Slices	5,164	33,280	15%
Number of Slices containing only related logic	5,164	5,164	100%
Number of Slices containing unrelated logic	0	5,164	0%
Total Number of 4 input LUTs	10,283	66,560	15%
Number used as logic	8,258		
Number used as a route-thru	2,025		
Number of bonded IOBs	516	633	81%
IOB Flip Flops	512		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 15,507 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	2,066	66,560	3%
Number of 4 input LUTs	8,254	66,560	12%
Logic Distribution			
Number of occupied Slices	5,161	33,280	15%
Number of Slices containing only related logic	5,161	5,161	100%
Number of Slices containing unrelated logic	0	5,161	0%
Total Number of 4 input LUTs	10,281	66,560	15%
Number used as logic	8,254		
Number used as a route-thru	2,027		
Number of bonded IOBs	516	633	81%
IOB Flip Flops	512		
Number of GCLKs	1	8	12%

4.3.5.2 Resultados ISE

De esta síntesis realizada en el programa ISE para el algoritmo B se obtiene los siguientes resultados resumidos en la siguiente tabla en función de los parámetros que son de interés para este estudio (flips-flops, slices, tamaño área (LUTs) y frecuencia máxima).

Los datos obtenidos de la síntesis con optimización en Area se resumen en la siguiente tabla.

	8 bits	16 bits	32 bits	64 bits	128 bits
Nº flip-flops	45	77	77	141	269
Nº slices	67	100	178	338	667
Nº LUTs	126	193	349	671	1312
Freq. Máx. (MHz)	135,707	135,543	94,885	82,034	63,524

	196 bits	256 bits	512 bits	1024 bits
Nº flip-flops	405	526	1039	2067
Nº slices	1035	1315	2596	5164
Nº LUTs	1990	2590	5156	10283
Freq. Máx. (MHz)	50,914	43,533	26,988	15,274

Estos datos se muestran en la siguiente gráfica resumen.

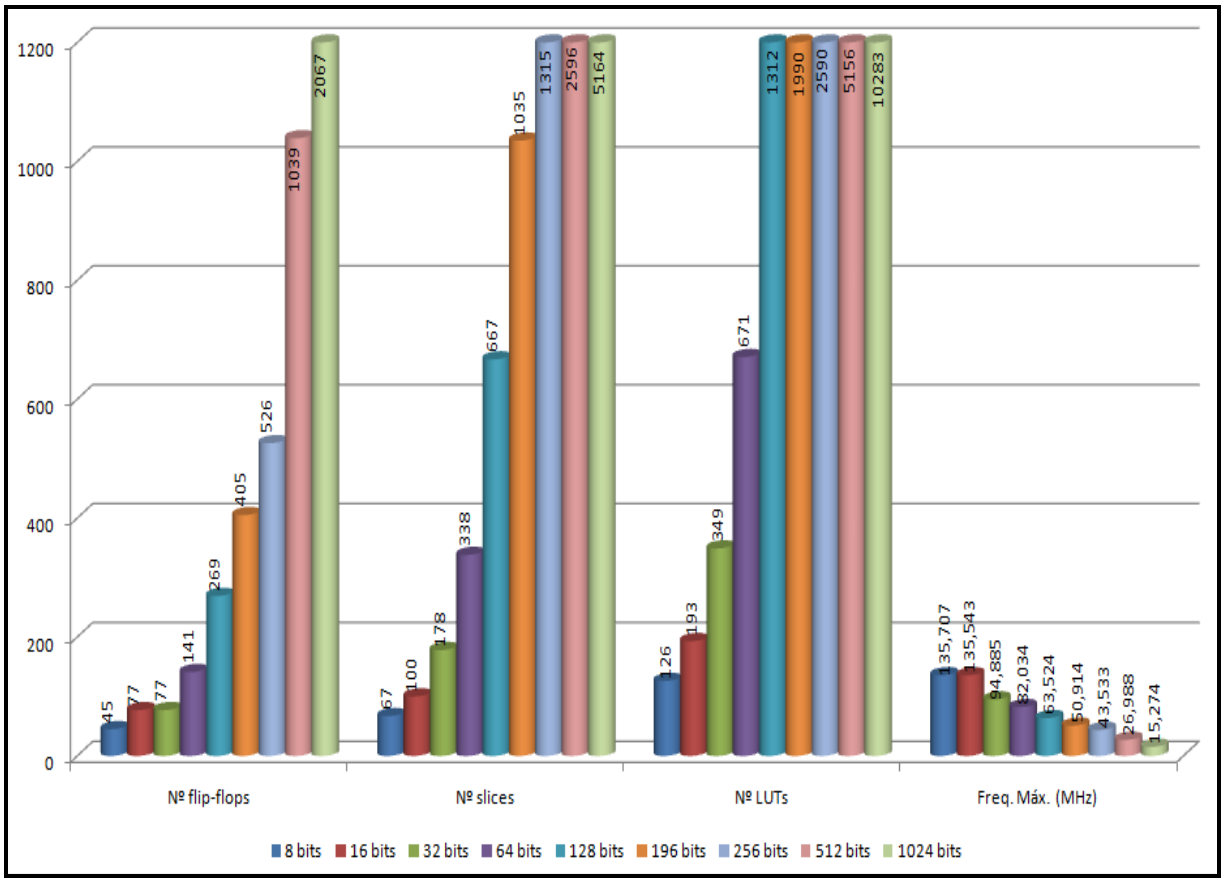

Figura 76.- Gráfica resumen síntesis ISE con optimización en Área del algoritmo B.

A continuación se muestra la tabla que recoge los datos optimizados en Tiempo.

	8 bits	16 bits	32 bits	64 bits	128 bits
Nº flip-flops	49	77	77	141	269
Nº slices	70	101	188	339	670
Nº LUTs	129	196	370	671	1311
Freq. Máx. (MHz)	177,099	136,412	106,055	89,245	67,764

	196 bits	256 bits	512 bits	1024 bits
Nº flip-flops	405	525	1039	2066
Nº slices	1033	1317	2595	5161
Nº LUTs	1990	2590	5154	10281
Freq. Máx. (MHz)	53,963	45,743	27,725	15,507

Se muestra la gráfica resumen de estos datos para el algoritmo B.

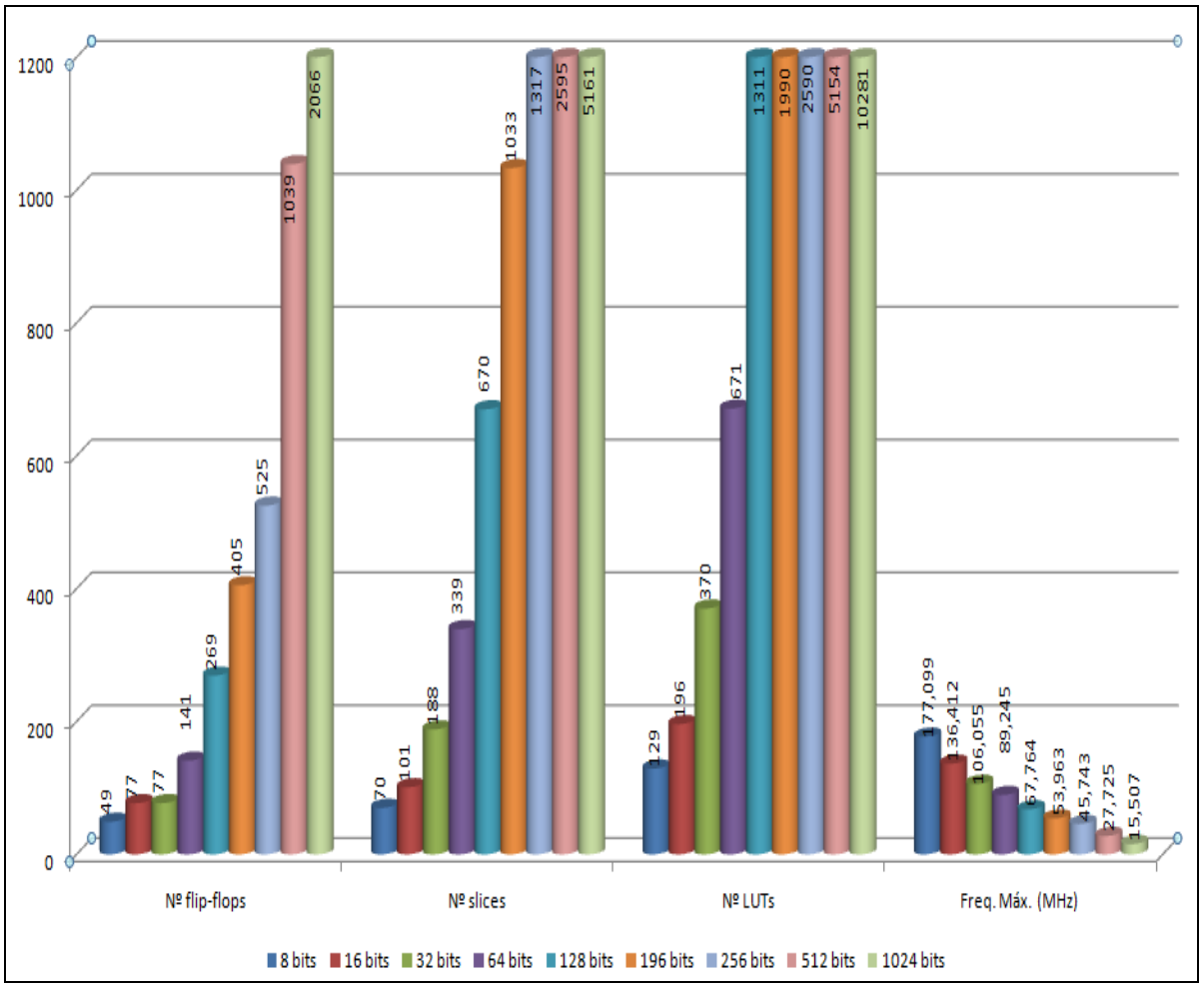

Figura 77.- Gráfica resumen síntesis ISE con optimización en Tiempo del algoritmo B.

4.3.5.3 Síntesis SYNOPSYS

Se realiza la síntesis con la herramienta Design Vision del programa Synopsys en la que se ha usado la opción de un esfuerzo alto en área a la hora de realizar esta síntesis, ya que para las especificaciones requeridas en este proyecto es necesario que se ocupe el menor área posible.

Para esta síntesis del algoritmo B se realiza también el estudio para los casos de 8, 16, 32, 64, 128, 196, 256, 512 y 1024 bits de entrada. En esta síntesis se extraen datos tanto de área como de consumo.

De los datos de área se obtendrá el número de *ports* que sería el número de puertos o entradas y salidas que se disponen en el diseño, el número de *nets* que sería el número de uniones o interconexionado, el número de *cells* que son el número total de las celdas o puertas lógicas usadas y el número de *references* que son los distintos tipos de puertas lógicas empleadas en la síntesis de este diseño. Por otra parte se dispondría del área perteneciente a la lógica combinacional y el área perteneciente a la lógica no combinacional además del *Net Interconnect area* que sería la parte de área ocupada por el interconexionado. Finalmente tendremos el *Total cell area* que sería el área total dispuesta para el cómputo total de las puertas lógicas, sumando la parte combinacional y la no combinacional; y si a este valor se le añade el área ocupada por el interconexionado se obtiene el *Total area.*

De los datos referidos al consumo se obtendrá la *Cell Internal Power* que se trata de la potencia consumida internamente por las celdas y la *Net Switching Power* que sería la potencia consumida asociada al interconexionado, que sumando estas dos potencias entre sí, se obtendría el consumo total dinámico para el diseño realizado ó *Total Dynamic Power*. A su vez también se obtendría el *Cell Leakage Power* que serían las pérdidas asociadas a este diseño.

8 bits de entrada -> N = 4 bits de salida

Datos de Área de la síntesis

```
Number of ports:                8
Number of nets:               249
Number of cells:              220
Number of references:          31

Combinational area:       2087.652987
Noncombinational area:    1315.115986
```

```
Net Interconnect area:     166.764182

Total cell area:          3402.768972
Total area:               3569.533155
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  = 191.1405 uW   (64%)
  Net Switching Power  = 108.2370 uW   (36%)
                         ---------
Total Dynamic Power    = 299.3775 uW  (100%)

Cell Leakage Power     =  12.5035 uW
```

16 bits de entrada -> N = 8 bits de salida

Datos de Área de la síntesis

```
Number of ports:              12
Number of nets:              419
Number of cells:             366
Number of references:         36

Combinational area:       3671.507972
Noncombinational area:    2210.903965
Net Interconnect area:     320.049533

Total cell area:          5882.411937
Total area:               6202.461470
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  = 330.2861 uW   (65%)
  Net Switching Power  = 180.7861 uW   (35%)
                         ---------
Total Dynamic Power    = 511.0722 uW  (100%)

Cell Leakage Power     =  21.4094 uW
```

32 bits de entrada -> N = 16 bits de salida

Datos de Área de la síntesis

```
Number of ports:              20
Number of nets:              578
```

```
Number of cells:              458
Number of references:          34

Combinational area:       6219.294913
Noncombinational area:    2409.967960
Net Interconnect area:     467.971269

Total cell area:          8629.262874
Total area:               9097.234143
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  = 611.2773 uW   (55%)
  Net Switching Power  = 500.6583 uW   (45%)
                         ---------
Total Dynamic Power    =   1.1119 mW  (100%)

Cell Leakage Power     =  31.0327 uW
```

64 bits de entrada -> N = 48 bits de salida

Datos de Área de la síntesis

```
Number of ports:              36
Number of nets:             1055
Number of cells:             839
Number of references:         32

Combinational area:       11728.996830
Noncombinational area:    4400.607914
Net Interconnect area:    1075.506894

Total cell area:          16129.604744
Total area:               17205.111638
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  = 757.0807 uW   (62%)
  Net Switching Power  = 468.8964 uW   (38%)
                         ---------
Total Dynamic Power    =   1.2260 mW  (100%)

Cell Leakage Power     =  56.1788 uW
```

128 bits de entrada -> N = 64 bits de salida

Datos de Área de la síntesis

```
Number of ports:              68
Number of nets:             2115
Number of cells:            1707
Number of references:         30

Combinational area:       23484.322673
Noncombinational area:    8381.887821
Net Interconnect area:    1956.859582

Total cell area:          31866.210495
Total area:               33823.070077
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   1.2569 mW   (66%)
  Net Switching Power  = 646.4268 uW   (34%)
                         ---------
Total Dynamic Power    =   1.9034 mW  (100%)

Cell Leakage Power     = 116.2726 uW
```

196 bits de entrada -> N = 98 bits de salida

Datos de Área de la síntesis

```
Number of ports:             102
Number of nets:             3212
Number of cells:            2600
Number of references:         29

Combinational area:       35752.571496
Noncombinational area:    12611.997723
Net Interconnect area:    3251.145428

Total cell area:          48364.569219
Total area:               51615.714646
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   1.6318 mW   (73%)
  Net Switching Power  = 614.7999 uW   (27%)
                         ---------
Total Dynamic Power    =   2.2466 mW  (100%)

Cell Leakage Power     = 174.7791 uW
```

256 bits de entrada -> N = 128 bits de salida

Datos de Área de la síntesis

```
Number of ports:              132
Number of nets:              4208
Number of cells:             3416
Number of references:          28

Combinational area:       46758.441350
Noncombinational area:    16344.447636
Net Interconnect area:    4273.383501

Total cell area:          63102.888986
Total area:               67376.272487
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   2.0565 mW   (72%)
  Net Switching Power  = 784.8051 uW   (28%)
                         ---------
Total Dynamic Power    =   2.8413 mW  (100%)

Cell Leakage Power     = 229.6417 uW
```

512 bits de entrada -> N = 256 bits de salida

Datos de Área de la síntesis

```
Number of ports:              260
Number of nets:              8530
Number of cells:             6970
Number of references:          27

Combinational area:       94194.418736
Noncombinational area:    32269.567265
Net Interconnect area:    9360.186968

Total cell area:          126463.986000
Total area:               135824.172968
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   3.9295 mW   (72%)
  Net Switching Power  =   1.5182 mW   (28%)
                         ---------
Total Dynamic Power    =   5.4478 mW  (100%)

Cell Leakage Power     = 469.5189 uW
```

1024 bits de entrada -> N = 512 bits de salida

Datos de Área de la síntesis

```
Number of ports:              516
Number of nets:             16840
Number of cells:            13744
Number of references:          33

Combinational area:       187596.530428
Noncombinational area:    64119.806522
Net Interconnect area:    23552.352236

Total cell area:          251716.336951
Total area:               275268.689187
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   7.8177 mW   (71%)
  Net Switching Power  =   3.1288 mW   (29%)
                         ---------
Total Dynamic Power    =  10.9464 mW  (100%)

Cell Leakage Power     = 932.5364 uW
```

4.3.5.4 Resultados SYNOPSYS

En la siguiente tabla resumen se muestran todos los resultados obtenidos de esta síntesis a través de la herramienta Design Vision de Synopsys, en la que se muestran los tipos de datos obtenidos en función del número de bits del diseño:

	8 bits	16 bits	32 bits	64 bits	128 bits
Ports	8	12	20	36	68
Nets	249	419	578	1055	2115
Cells	220	366	458	839	1707
References	31	36	34	32	30
Comb. Area (µm²)	2.088	3.672	6.219	11.729	23.484
No comb. Area (µm²)	1.315	2.211	2.410	4.401	8.382
Net area (µm²)	167	320	468	1.076	1.957
Total cell area (µm²)	3.403	5.882	8.629	16.130	31.866
Total area (µm²)	3.570	6.202	9.097	17.205	33.823
Consumo total (mW)	0,30	0,51	1,11	1,23	1,90
Pérdidas (µW)	12,50	21,41	31,03	56,18	116,27

	196 bits	256 bits	512 bits	1024 bits
Ports	102	132	260	516
Nets	3212	4208	8530	16840
Cells	2600	3416	6970	13744
References	29	28	27	33
Comb. Area (µm²)	35.753	46.758	94.194	187.597
No comb. Area (µm²)	12.612	16.344	32.270	64.120
Net area (µm²)	3.251	4.273	9.360	23.552
Total cell area (µm²)	48.365	63.103	126.464	251.716
Total area (µm²)	51.616	67.376	135.824	275.269
Consumo total (mW)	2,25	2,84	5,45	10,95
Pérdidas (µW)	174,78	229,64	469,52	932,54

A continuación se muestra una gráfica con el número de puertas lógicas empleadas en cada diseño según los bits empleados a la entrada.

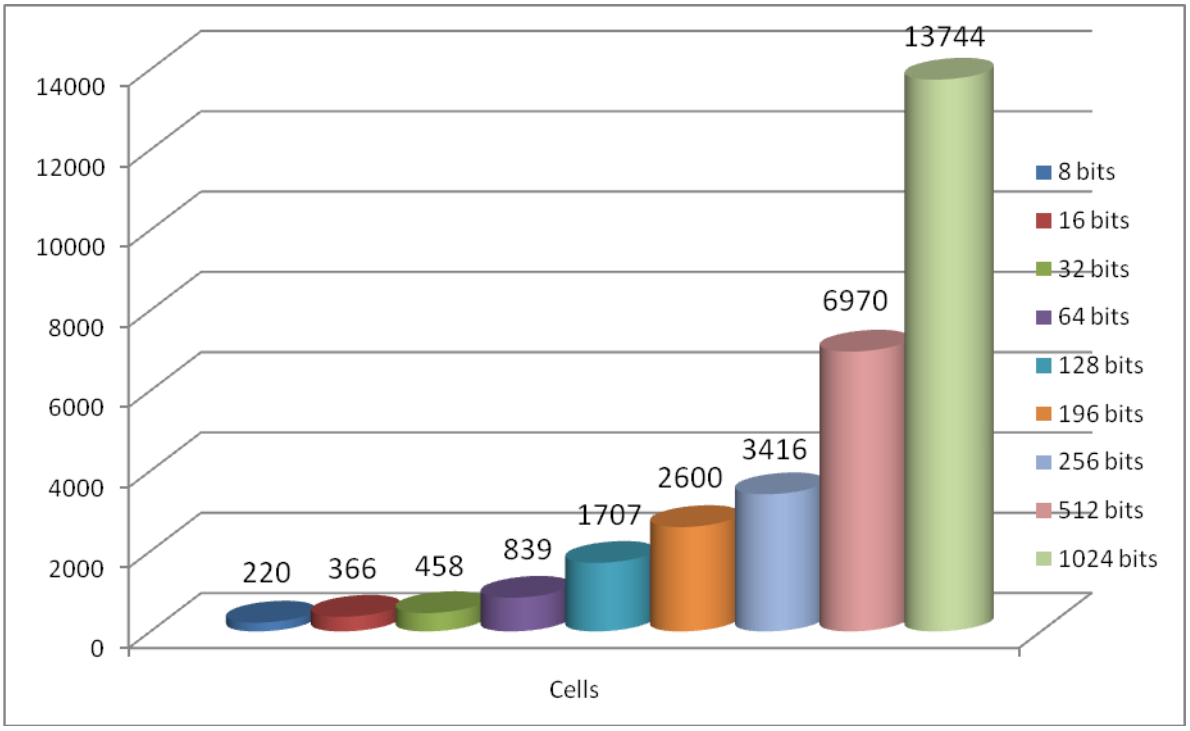

Figura 78.- Gráfica puertas lógicas algoritmo B.

En la siguiente gráfica se muestra el total del área según el número de bits empleados a la entrada con sus respectivas particiones en área correspondiente a la parte de la lógica combinacional y a la parte de la lógica no combinacional. Cabe destacar que para el total del área a parte de sumar estas dos partes correspondientes a la lógica, habría que añadirle la parte correspondiente al área del interconexionado.

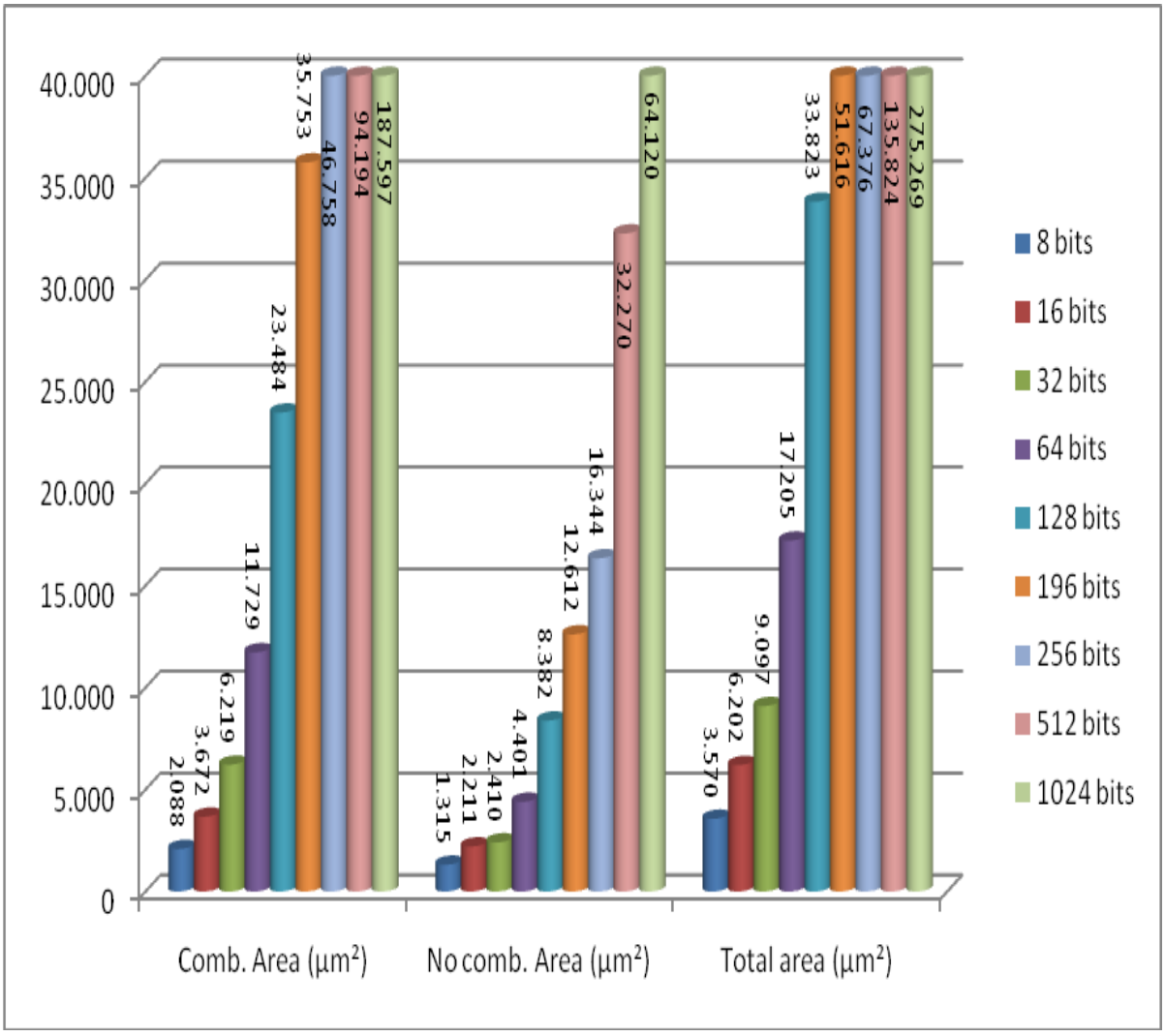

Figura 79.- Gráfica repartición área algoritmo B.

Se muestra la potencia que se consumiría (en unidades de miliwatios) por cada diseño en función del número de bits que se utilicen a la entrada por medio de la siguiente gráfica:

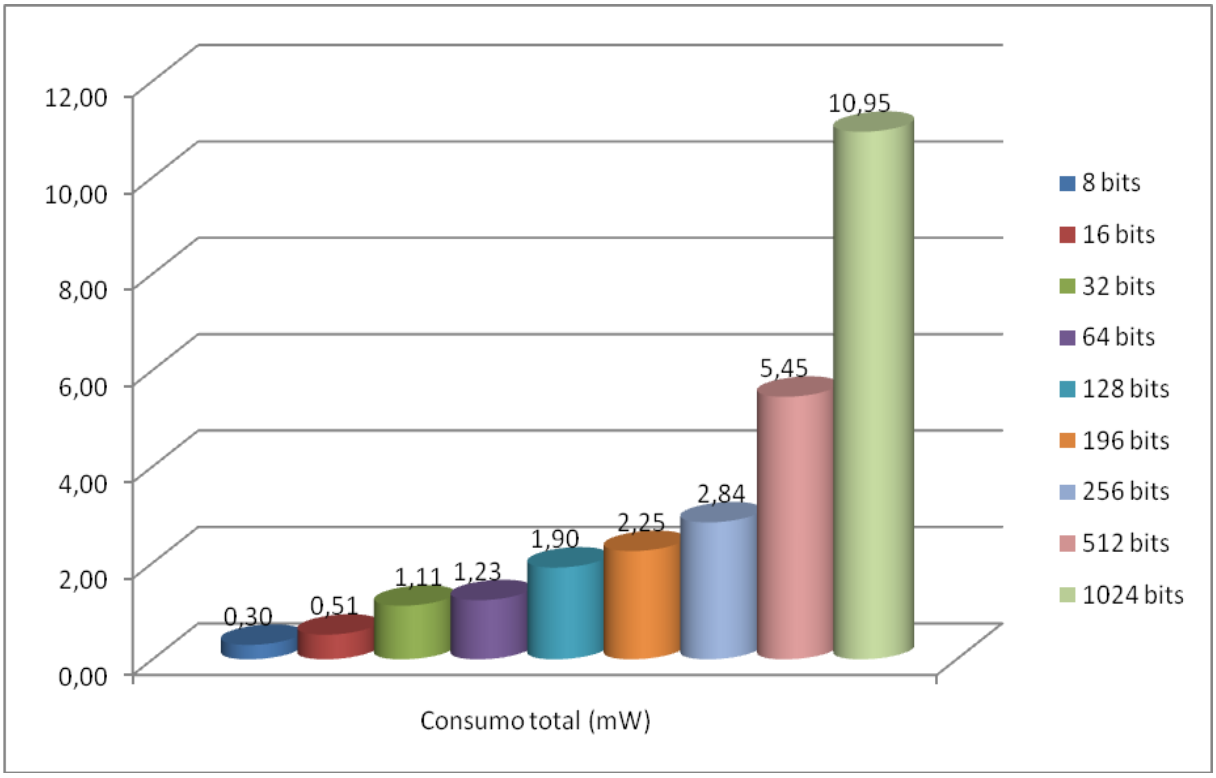

Figura 80.- Gráfica consumo algoritmo B.

4.3.6 Conclusiones

Una vez realizado el diseño de este generador de números pseudo-aleatorios para el algoritmo B en el lenguaje de descripción hardware vhdl y sus posteriores implementaciones tanto para la tecnología de dispositivos programables FPGAs como para la tecnología de circuitos a medida ASICs, se pueden analizar los resultados obtenidos y llegar a una serie de conclusiones a partir de los mismos.

Destacar que mediante el ejecutable creado en Turbo C++, hemos comprobado que el diseño estaba realizado correctamente y posee un funcionamiento correcto de acuerdo a lo esperado.

De los resultados obtenidos en la implementación de dispositivos programables a partir del programa ISE, al igual que tanto para el generador Blum Blum Shub como para el generador anterior del algoritmo A se puede seguir observando que comparando las dos tablas de los dos tipos de optimizaciones llevadas a cabo, en el caso de la optimización efectuada con un

alto esfuerzo en área se sigue consiguiendo reducir ligeramente el tamaño del área medida en LUTs, así como el número de flip-flops y el número de slices respecto a la optimización con un alto esfuerzo en requerimientos de tiempo (aunque bien es cierto que para este algoritmo B las diferencias son menores); al igual, con la optimización en tiempo se sigue consiguiendo una frecuencia máxima ligeramente mayor aumentando así la velocidad.

Nuevamente si para esta misma síntesis de dispositivos programables se compara ahora en función del número de bits a tratar en el proceso, se comprueba que también al igual que en el generador Blum Blum Shub y en el algoritmo A a medida que aumenta el número de bits se va incrementando tanto el tamaño del área como el número de flip-flops y slices (figuras 76 y 77), también a cambio, se observa que a medida que se aumenta el número de bits a tratar se reduce sustancialmente la frecuencia máxima haciendo los procesos cada vez más lentos.

En cuanto a la síntesis realizada para la tecnología de las ASICs también se extrae la conclusión de que a medida que se aumenta el número de bits a tratar se incrementan las puertas lógicas, el interconexionado, el tamaño del área de estas celdas, ya sea en la parte correspondiente a la lógica combinacional como a la parte correspondiente a la lógica no combinacional.

De la misma manera, el estudio realizado del consumo de la potencia medida en miliwatios nos aporta que se consume bastante más a medida que aumenta el número de bits para estos generadores de números aleatorios, lo mismo que ocurre con las pérdidas asociadas a esta potencia aunque en magnitudes mucho más inferiores como ya habíamos mencionado.

DISEÑO PRNG´S

(PSEUDO-RANDOM NUMBER GENERATION)

OPCIÓN C

ALGORITMO – IMPLEMENTACIÓN VHDL – SIMULACIÓN – COMPROBACIÓN LENGUAJE C – SÍNTESIS ISE Y SYNOPSYS – RESULTADOS – CONCLUSIONES

4.4 Opción C

4.4.1 Algoritmo

Para este apartado se va a estudiar el tercero de los algoritmos de los generadores de números pseudo-aleatorios que se quieren implementar en la FPGA para usarse en la tecnología RFID, este algoritmo al igual que los dos anteriores también es ultraligero y está basado en el uso de una función triangular.

Se muestra el correspondiente código C de este algoritmo que se usó previamente para observar la calidad de la salida en la búsqueda de la obtención de los números aleatorios.

```
Opción C

   x[0]= x[0] + ((x[0]*x[0])|5);
   x[1]= x[1] + ((x[1]*x[1])|13);

//Non-linerar function

r1=32/ se podría bajar hasta 24 rondas


z[0]=x[0];


for(i=0;i<32;i++){z[0]=(z[0]<<1)+((z[0]+(0x56AB0A))>>1);
 y[0]=z[0];
 y[0]=(y[0]>>1)+(y[0]<<1)+y[0]+(x[1]);
 }



z[0]=z[0]^y[0];


//Filter output
// z[0] es la salida del generador
// me quedo con los bits menos significativos que son los que varían mas
rápidamente
// para el ejemplo de variables de 32 bits me quedo con los 16 menos
significativos


 z[0]=z[0]&0x0000ffff;
```

Para este algoritmo se realizará el estudio para los mismos casos de número de bits a la entrada que para los dos algoritmos anteriores A y B.

Este algoritmo C también parte con las dos primeras sentencias iguales a las de los algoritmos A y B por lo que al introducir el valor semilla realizará el mismo tipo de operaciones que en A y B y se obtendrán los valores de las constantes x[0] y x[1].

La constante r1 es el valor que se le asigna al contador para definir el número de veces que itera el proceso para realizar las operaciones establecidas en un bucle. Este valor se establece en 32 para obtener una mayor seguridad a la hora de que esos números generados sean todavía más aleatorios, pero en el estudio previo realizado sobre la calidad de la salida de estos números se comprobó que este valor se podría reducir hasta 24 para que los números generados siguiesen siendo suficientemente aleatorios para conseguir un seguro código encriptado.

Se va a realizar un bucle que se repite el número de veces que indique r1. En primer lugar se le asigna a la señal z el valor de la constante x[0]. Dentro de las operaciones a realizar por el bucle primero se haría un registro de desplazamiento a la izquierda de la señal z junto a un registro de desplazamiento a la derecha de la suma de la señal z más el número hexadecimal 56AB0A, seguidamente estas operaciones se suman y se asigna a la señal y[0] ese valor. Como este proceso se realiza consecutivamente mientras se realiza el bucle, lo que se haga en las r1 menos 1 veces en la señal y[0] se está obviando cuando se realiza esta asignación.

Entonces será en el momento en que el bucle se esté realizando por la r1 vez cuando a ese valor asignado de la señal z a la señal y[0] se le realiza un desplazamiento a la derecha, un desplazamiento a la izquierda y estos valores se suman junto al propio valor de y[0] y a la constante x[1] para obtener así el valor final de la señal y[0].

A los dos valores finales de z[0] e y[0] de después de la ejecución del bucle se les realiza un XOR a nivel de bit a bit[8] y éste valor obtenido se asigna en z[0].

Por último se le aplica el filtro de salida a la señal z para quedarse con el número de bits deseados a la salida en función de los bits de la entrada según el caso de los estudiados que se esté tratando en ese momento, así se obtendría el valor de z[0] quedándose, al igual que en los algoritmos A y B, con los bits

[8] Véase Anexo B

menos significativos (los bits de la derecha) que son los que varían más rápidamente y son los que interesan en una segura encriptación siendo así lo más aleatorios posibles.

4.4.2 Implementación en código Vhdl

La implementación de este algoritmo C en vhdl se ha llevado a cabo mediante la creación de una máquina de estados. Dicha implementación se rige por la siguiente entidad:

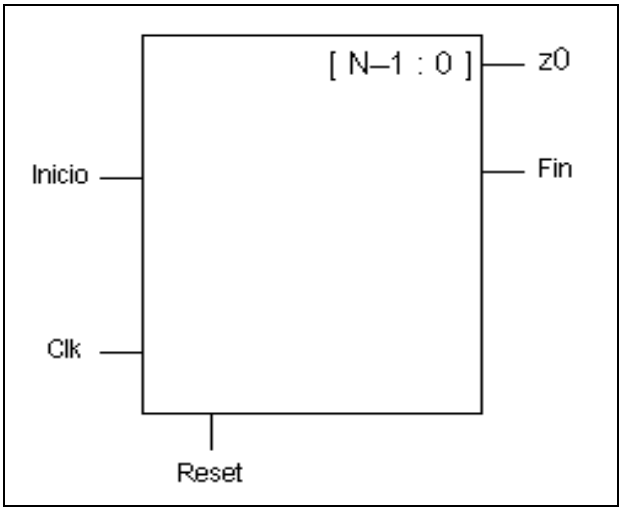

Figura 81.-Entidad algoritmo C.

De esta entidad al igual que en los dos algoritmos anteriores A y B, se observa que no hay señales de entrada específicas para la introducción de los datos o variables de entrada (bits que vamos a emplear para generar los números aleatorios), por lo que también emplearemos las semillas como constantes, ya que así no seran sintetizadas en el diseño. Estas constantes que son las que serán introducidas como valores semilla seed_0 y seed_1 se regirán por los mismos valores que los empleados en los algoritmos A y B, y como las dos primeras sentencias son comunes a los tres algoritmos, las constantes x0 y x1 que se calculan a partir de estas semillas también serán iguales que en estos algoritmos A y B.

La máquina de estados empleada para la implementación de este algoritmo C en vhdl sería la siguiente:

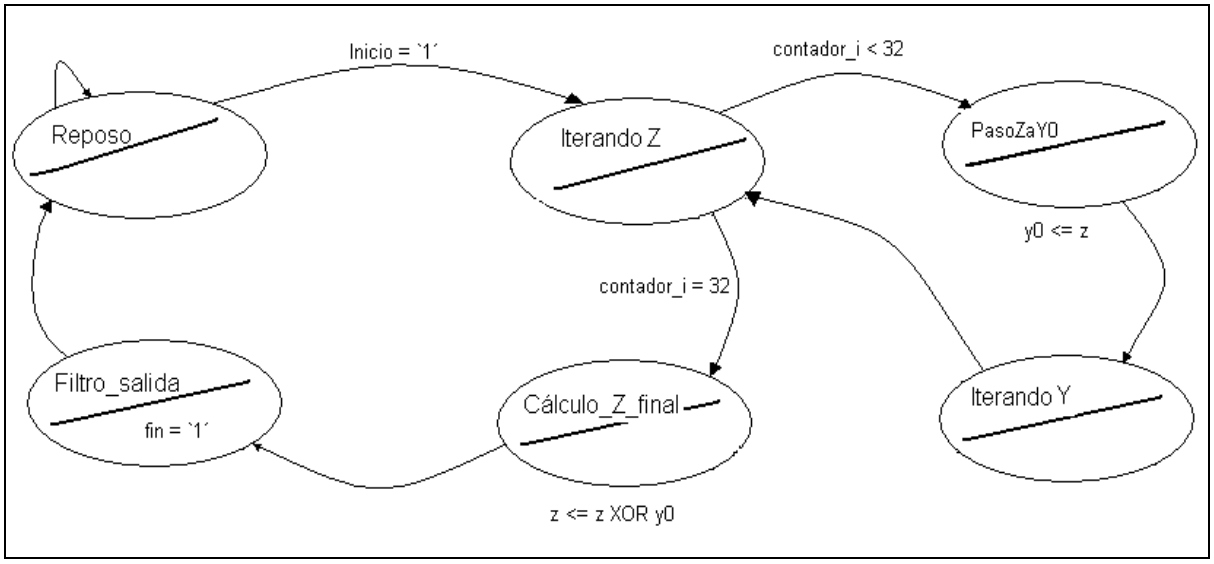

Figura 82.- Diagrama de estados algoritmo C.

a) REPOSO: Está a la espera del bit de Inicio para comenzar el proceso.

   Si es 1 go to Iterando, sino mantiene estado

b) ITERANDO Z: Opera con la señal z.

   Incrementa contador_i

   Si contador_i = 32 go to Cálculo_Z_final, sino go to PasoZay0

c) PASOzAy0: Asigna el valor de la señal z a la señal y0

   Go to Iterando Y

d) ITERANDO Y: Opera con la señal y0.

   Go to Iterando Z

e) CÁLCULO_Z_FINAL: Calcula el valor final de la señal z

   Go to Filtro_salida

f) FILTRO_SALIDA: Se queda con los bits menos significativos.

   Activa la salida fin

   Go to Reposo

Al igual que en algoritmo B en esta implementación no es posible realizar los registros de desplazamiento de manera manual como se hacía en el algoritmo A, sino que es necesario usar los comandos shift_left y shift_right, por lo que también se han debido de cambiar las bibliotecas (`std_logic_unsigned` y `std_logic_arith`) por la biblioteca (`numeric_std`). A su vez también hay que tener en cuenta que a la hora de realizar el diseño vhdl con esta nueva biblioteca para definir las señales hay que usar unsigned en lugar de std_logic_vector y que no se puede usar la función conv_std_logic_vector sino que se debe introducir el número directamente como hexadecimal (x``……..´´).

La otra consideración que también hay que tener en cuenta a la hora de realizar el diseño es que en este algoritmo también se opera con el número hexadecimal 56AB0A, por lo que se creará también un estado auxiliar con una señal auxiliar z_AUX del mismo tamaño que el número hexadecimal (24 bits) para poder operar con él en el estado Iterando Z y luego truncar en el estado Auxiliar la señal z ajustándola al tamaño de bits de la entrada.

Para ello se tiene la siguiente máquina de estados:

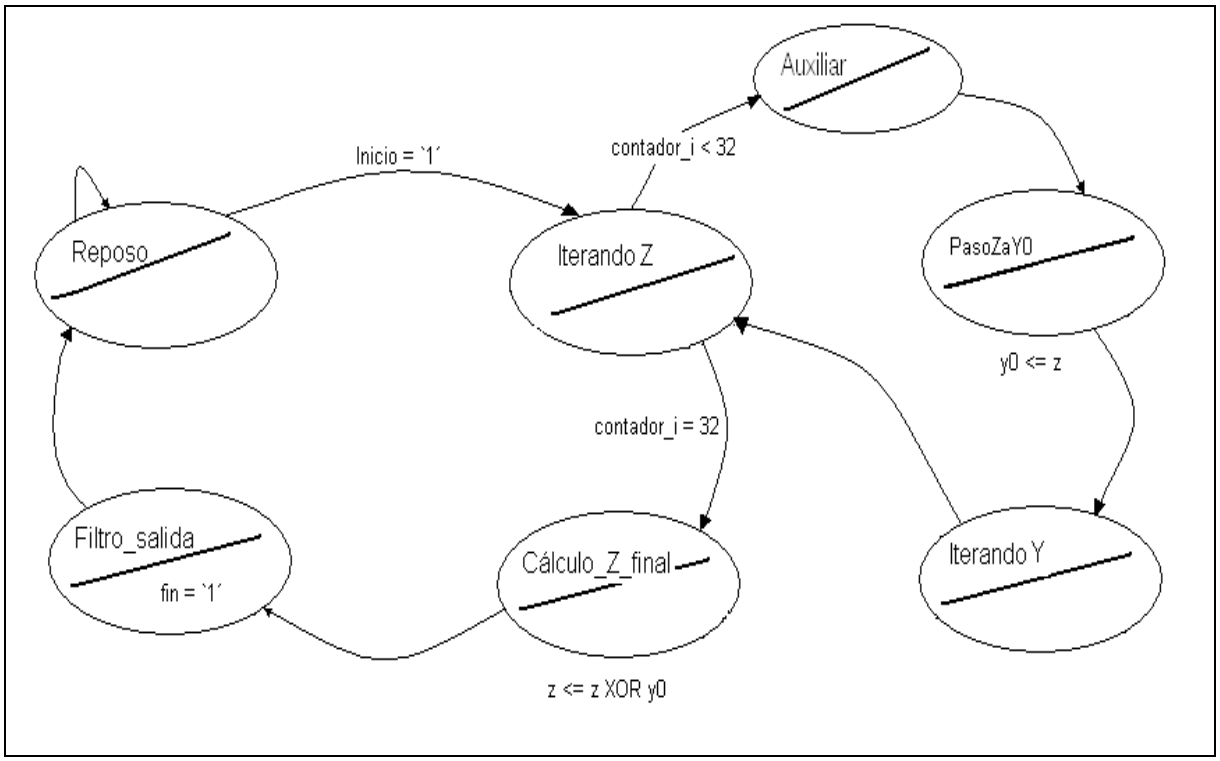

Figura 83.- Diagrama de estados para generadores de 8 y 16 bits de entrada para el algoritmo C.

a) REPOSO: Está a la espera del bit de Inicio para comenzar el proceso.

Si es 1 go to Iterando, sino mantiene estado

b) ITERANDO Z: Opera con la señal z.

Incrementa contador_i

Si contador_i = 32 go to Cálculo_Z_final, sino go to Auxiliar

c) AUXILIAR: Se trunca el valor obtenido de las operaciones de la señal z y el número hexadecimal.

Go to PasoZay0

d) PASOzAy0: Asigna el valor de la señal z a la señal y0

Go to Iterando Y

e) ITERANDO Y: Opera con la señal y0.

Go to Iterando Z

f) CÁLCULO_Z_FINAL: Calcula el valor final de la señal z

Go to Filtro_salida

g) FILTRO_SALIDA: Se queda con los bits menos significativos.

Activa la salida fin

Go to Reposo

El código empleado en la implementación de este algoritmo C se muestra en el apartado 4 del anexo A, en él se muestra tanto el ejemplo para 32 bits de entrada como el ejemplo de 16 bits de entrada con el estado Auxiliar. Para los demás casos mayores de 32 bits de entrada valdría con cambiar el valor del genérico N (que está puesto de tal modo que sea equivalente al nº de bits de salida) en el ejemplo de 32 bits, y para el caso de 8 bits de entrada habría que cambiar este genérico N en el ejemplo de 16 bits.

De la arquitectura empleada en la implementación de este algoritmo C se obtiene el siguiente esquema de componentes:

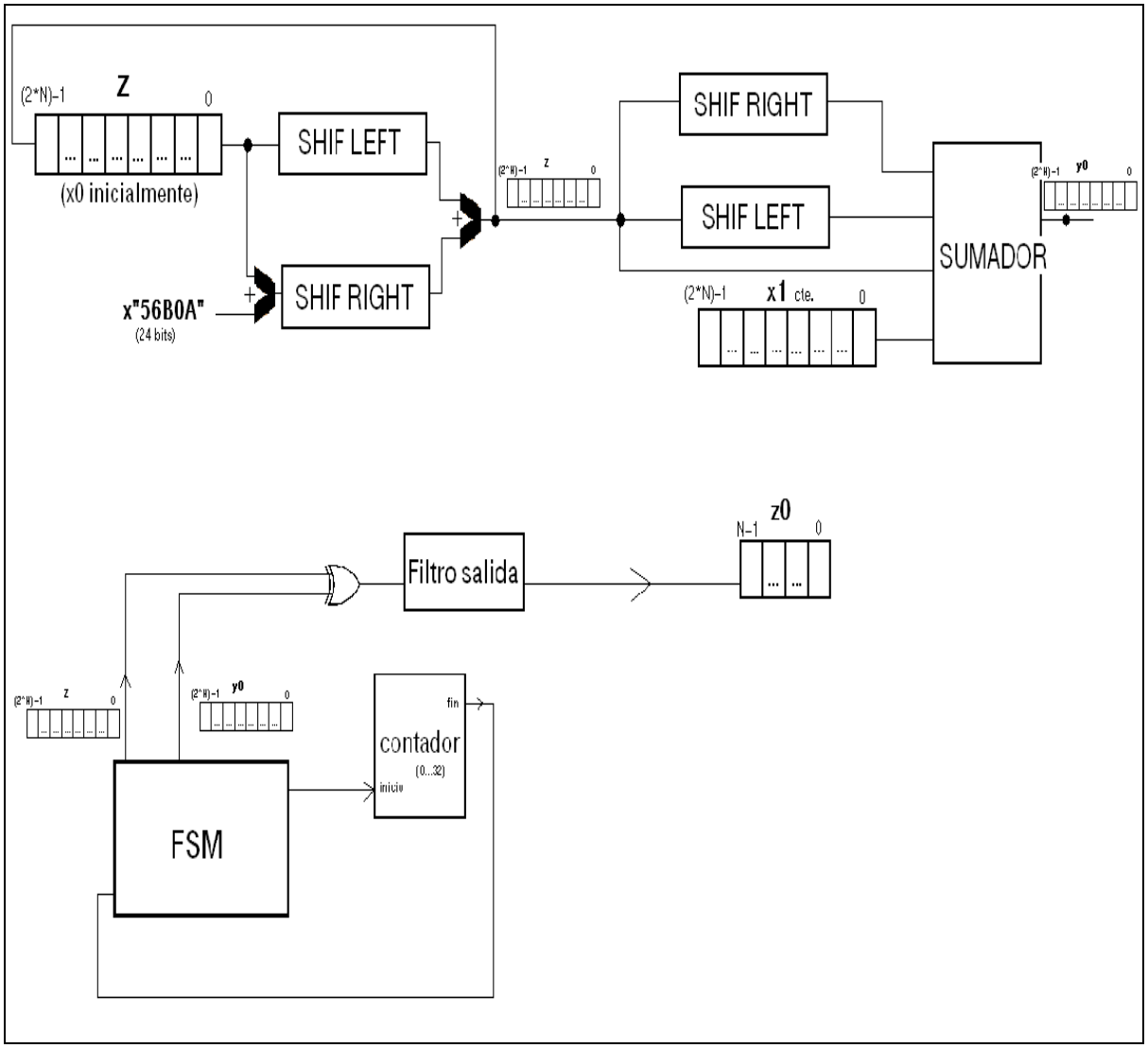

Figura 84.- Esquema de componentes algoritmo C.

De este esquema de componentes del algoritmo C se observa que hay un control (FSM) que gestiona el proceso mediante el uso de un contador que va de 0 a 32, y que una vez realizado todo el proceso, el control envía como salida las señales z e y0 que son operadas por un XOR para posteriormente filtrarse y obtener la salida z0.

El proceso gestionado por la FSM contiene un sumador de cuatro entradas, otros dos sumadores de dos entradas, dos registros de desplazamiento a la izquierda, dos registros de desplazamiento a la derecha, a su vez también se observa el empleo de las constantes x0 y x1 y del número hexadecimal 56AB0A.

4.4.3 Simulación

En este apartado se procede a la simulación de este algoritmo C, donde en primer lugar se observará a modo de ejemplo la simulación para 32 bits de entrada que genera un número aleatorio en z0.

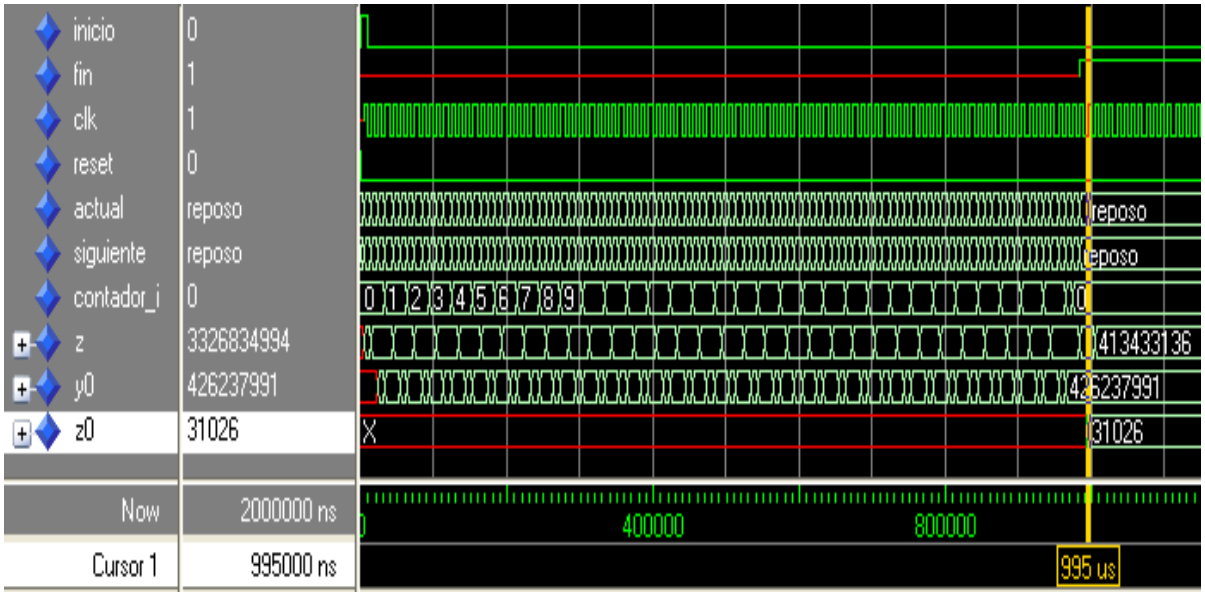

Figura 85.- Simulación 1 algoritmo C.

En la siguiente simulación se observa como el bit de inicio activa el comienzo del algoritmo, y como el algoritmo se va ejecutando a través de los tres estados Iterando z, Paso_z_a_y0 e Iterando Y que corresponderían al for del código C de partida. A su vez también se observa como cada vez que se da el estado Paso_z_a_y0 se elimina el valor anterior que tuviese la señal y0 para establecer el nuevo valor de la señal z, esto hace que no tenga memoria la señal y0 porque no vuelve a usar esos valores calculados en cada ejecución del estado Iterando Y salvo cuando se esté ejecutando por última vez.

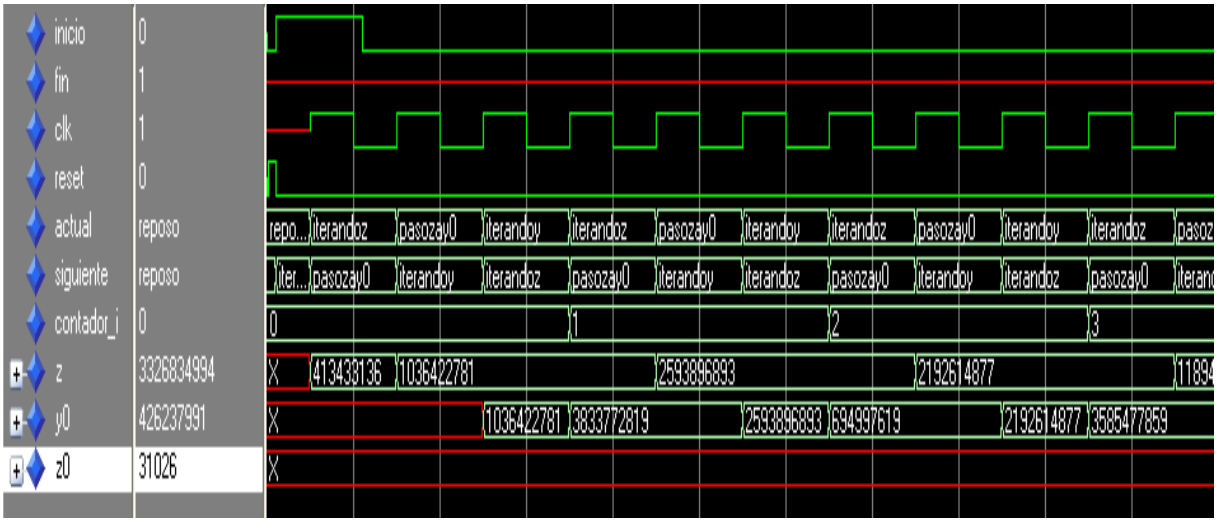

Figura 86.- Simulación 2 algoritmo C.

Aquí se observa como el contador_i llega a su fin en 32 y es cuando no se elimina ese último valor obtenido en la señal y0 y se opera con él.

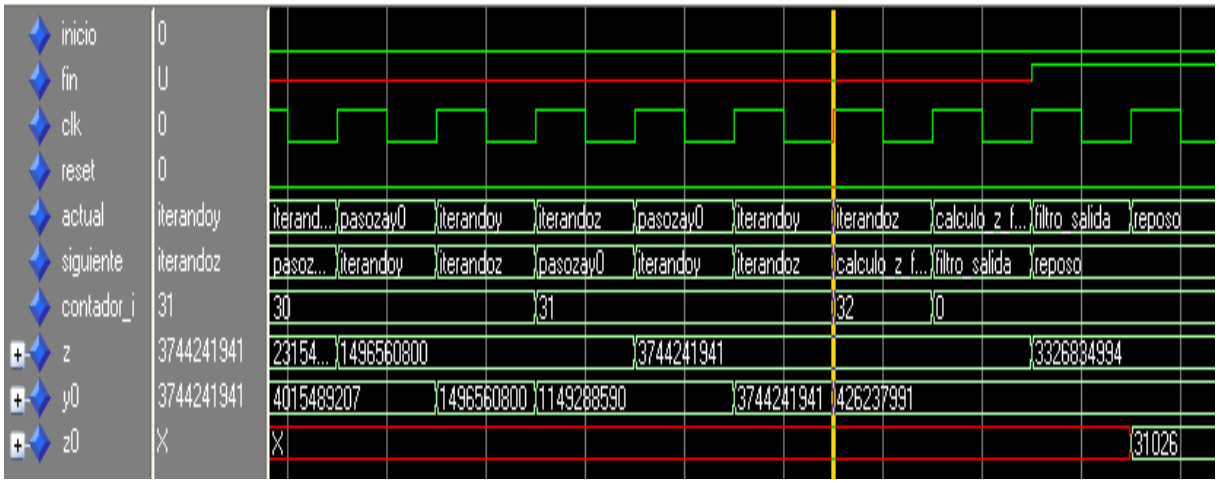

Figura 87.- Simulación 3 algoritmo C.

En binario se observa mejor como este valor obtenido en la señal y0 opera en el estado Cálculo_final por medio del XOR junto al valor obtenido de la señal z. El resultado de este XOR se almacena nuevamente en la señal z, al cual se le aplica el filtro de salida quedándose también reflejado así de mejor manera en binario.

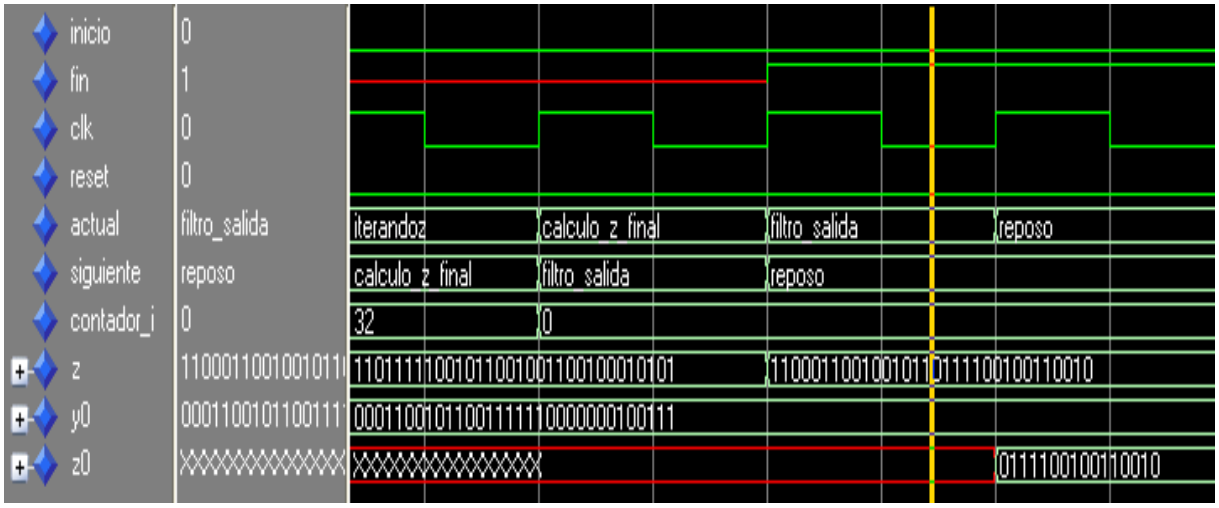

Figura 88.- Simulación 4 algoritmo C.

Ahora se simula exhaustivamente uno de los casos problemáticos como es el caso de 16 bits de entrada para el que es necesario el empleo del estado Auxiliar para poder operar con el número hexadecimal 56AB0A que es mayor que las señales a tratar, puesto que estas señales z e y0 son de 16 bits (para este caso) mientras que este número hexadecimal es de 24 bits. En primer lugar se tiene la simulación completa en la que se observa lo que tarda en realizarse el algoritmo completo y el número generado para las semillas introducidas como ejemplo.

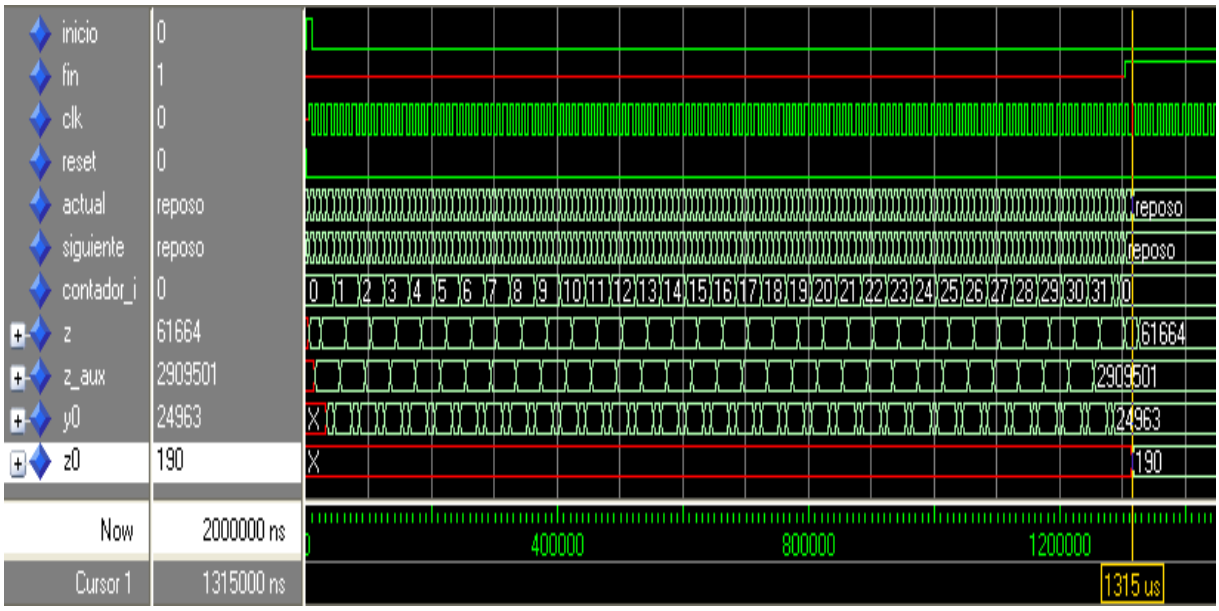

Figura 89.- Simulación 5 algoritmo C.

En la siguiente simulación se muestra como se hace uso del estado Auxiliar para solucionar este problema.

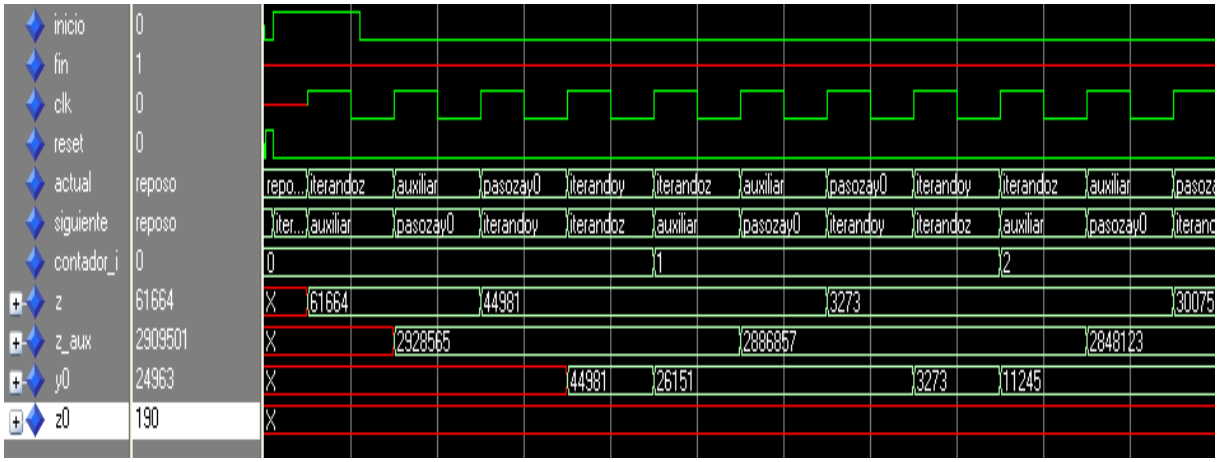

Figura 90.- Simulación 6 algoritmo C.

La simulación para el caso de 8 bits de entrada sería como la observada para el caso de 16 bits, se simula para ver el tiempo de ejecución y el número generado a la salida z0.

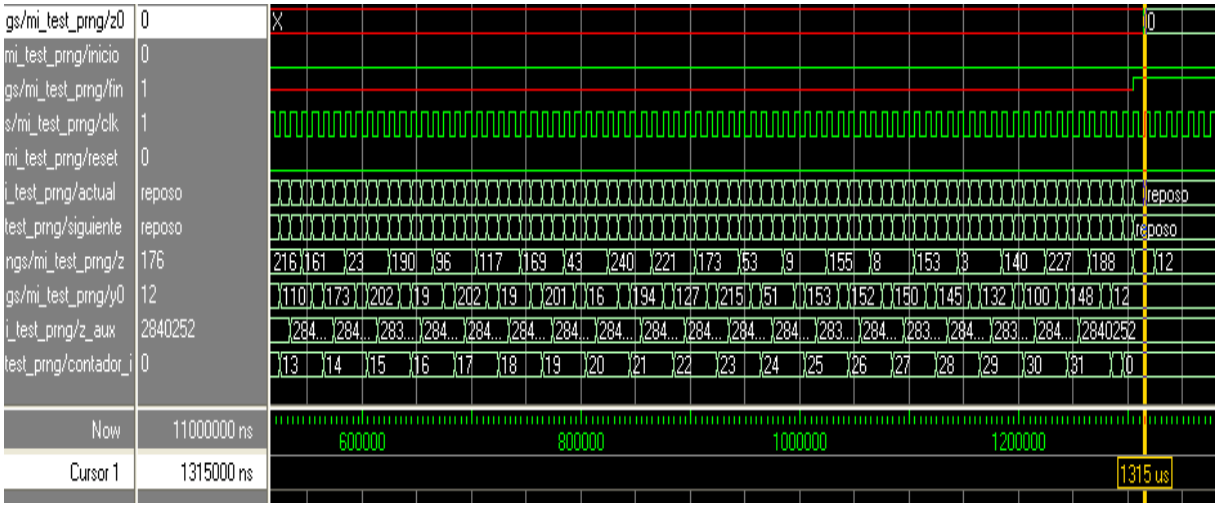

Figura 91.- Simulación 7 algoritmo C.

En las simulaciones posteriores vamos a observar los resultados y el tiempo de ejecución de los casos mayores de 32 bits de entrada que tienen un proceso análogo a éste.

Primero el caso de 64 bits de entrada que genera un número pseudo-aleatorio de 48 bits en la salida z0.

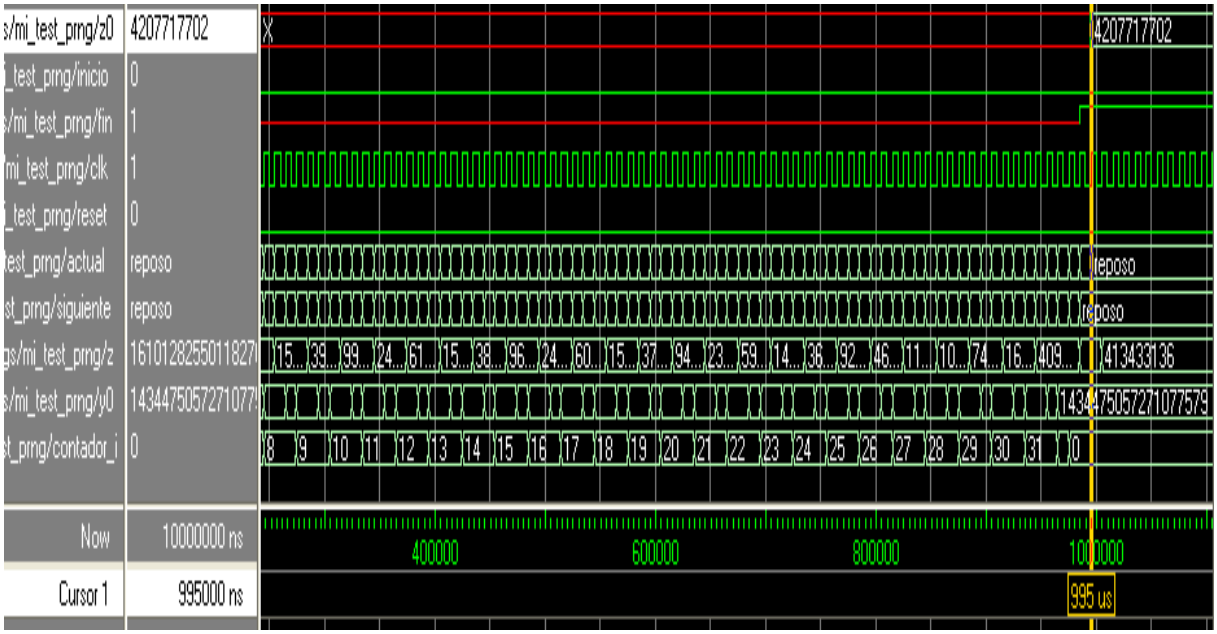

Figura 92.- Simulación 8 algoritmo C.

Simulación para 128 bits de entrada => 64 bits de salida z0. Al igual que en el algoritmo B, según vamos aumentando el número de bits hay que tener en cuenta una consideración muy importante, y es que al usar una biblioteca distinta (`numeric_std`) en el diseño vhdl y no poder utilizar la función conv_std_logic_vector, al definir las constantes como números decimales hay que añadir ceros a la izquierda del número hexadecimal para adaptarlo al tamaño de cada caso de simulación que se realice según su número de bits a la entrada.

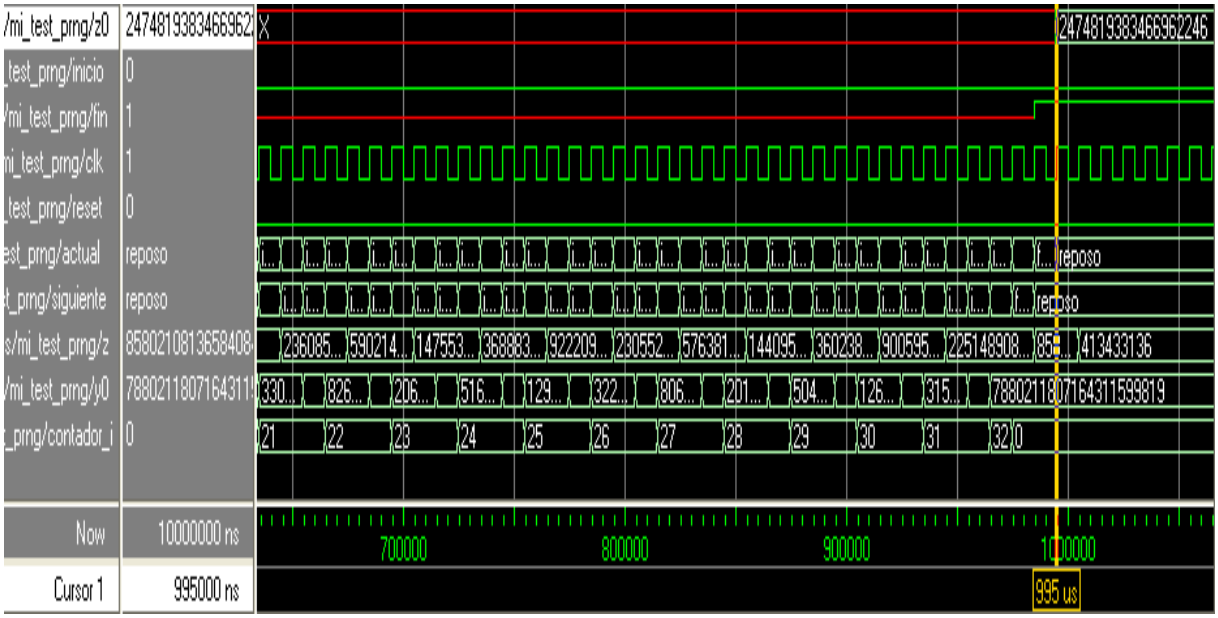

Figura 93.- Simulación 9 algoritmo C.

Simulación para 196 bits de entrada => 98 bits de salida z0.

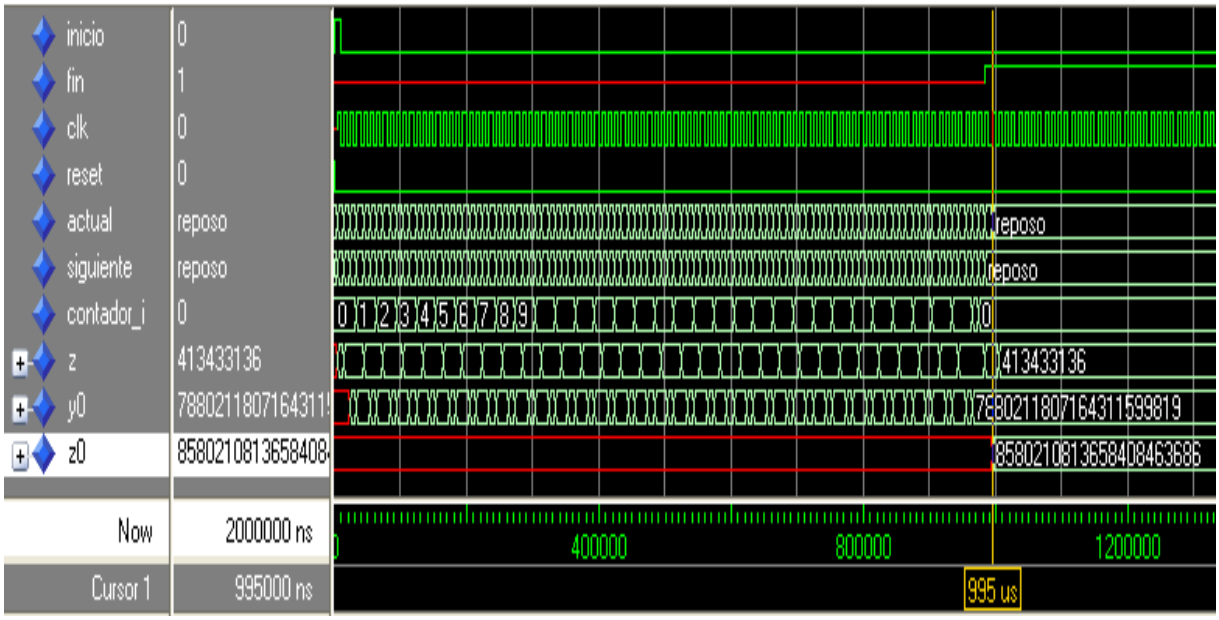

Figura 94.- Simulación 10 algoritmo C.

Simulación para 256 bits de entrada => N = 128 bits de salida z0.

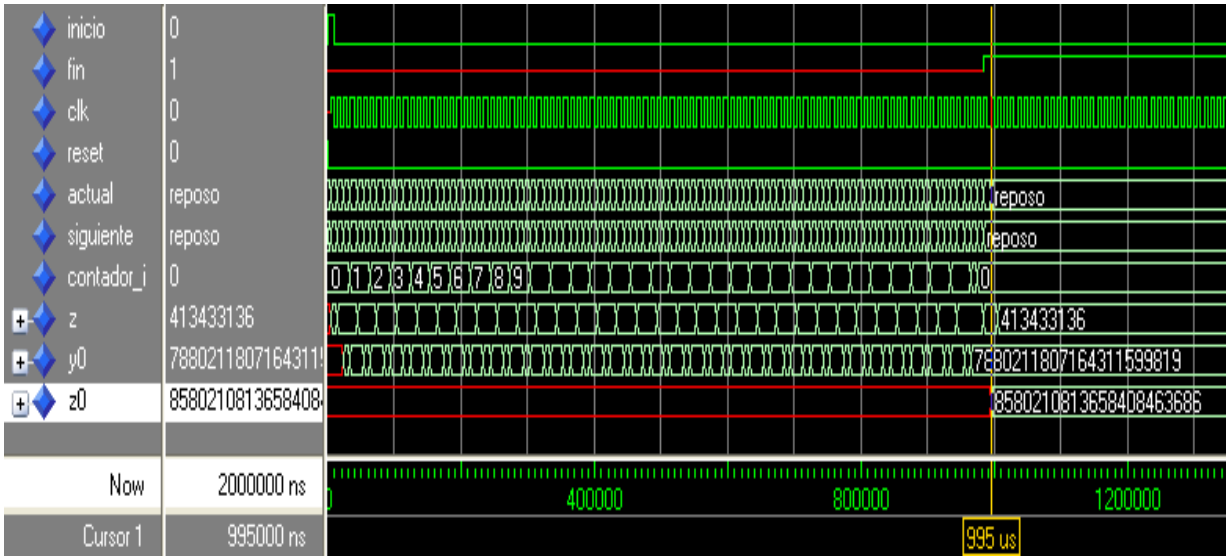

Figura 95.- Simulación 11 algoritmo C.

Simulación para 512 bits de entrada => N = 256 bits de salida z0.

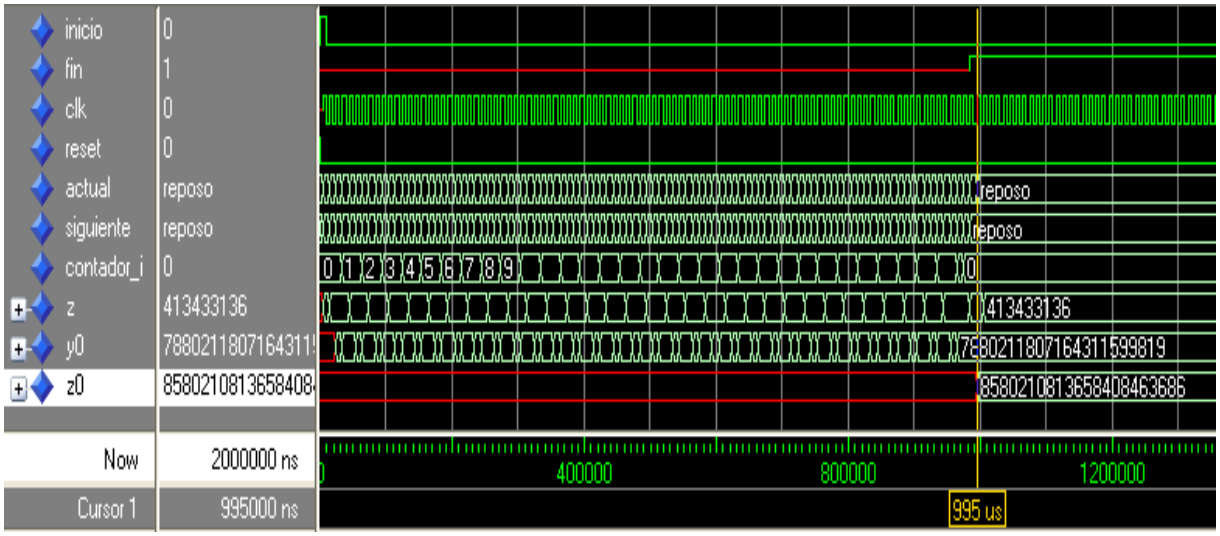

Figura 96.- Simulación 12 algoritmo C.

Simulación de 1024 bits de entrada => N = 512 bits de salida z0.

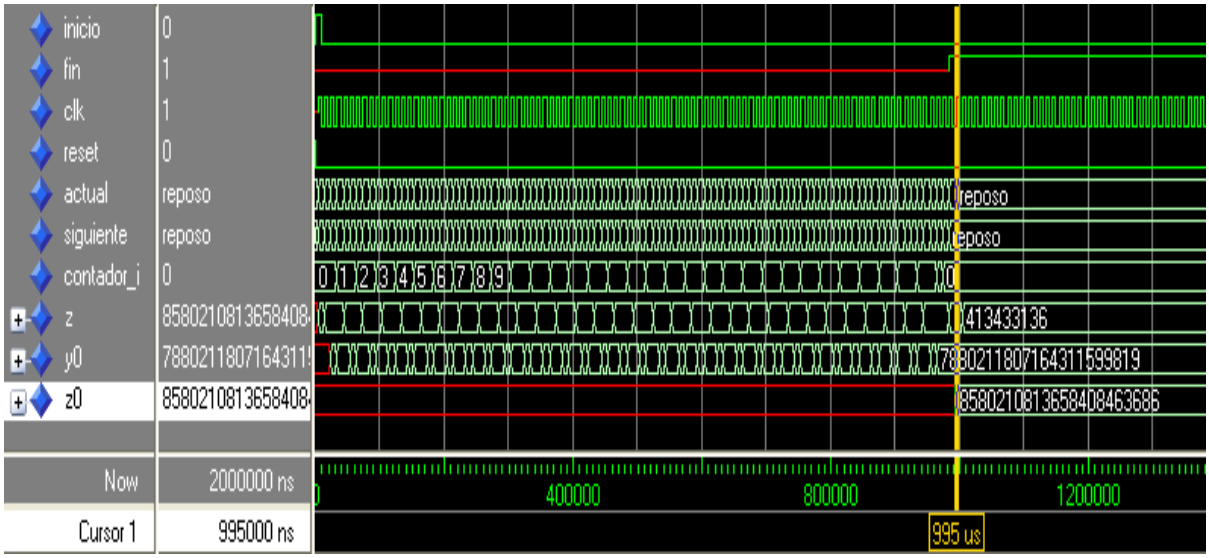

Figura 97.- Simulación 13 algoritmo C.

De estas simulaciones se comprueba que para los casos de 32 bits y mayores se emplea el mismo tiempo de ejecución del algoritmo que es de 995 µs. Para los casos problemáticos de 8 y 16 bits de entrada se emplean 1315 µs un tiempo mayor como es lógico, ya que emplea un estado más con el auxiliar y las operaciones que conlleva.

A su vez de estas simulaciones se extraen los siguientes resultados como números generados en la salida z0.

Nº de bits de entrada	Genérico N de bits	Nº de bits de la salida z0	Nº generado en la salida z0
8	4	4	0
16	8	8	190
32	16	16	31.026
64	32	48	4.207.717.702
128	64	64	2.474.819.383.466.962.246
196	98	98	8.580.210.813.658.408.463.686
256	128	128	8.580.210.813.658.408.463.686
512	256	256	8.580.210.813.658.408.463.686
1024	512	512	8.580.210.813.658.408.463.686

Al igual que en los dos algoritmos anteriores A y B se observa que el resultado se repite para los casos de un número de bits muy elevado a la entrada,

puesto que para las constantes no se permiten introducir números tan elevados en vhdl; el número entero más elevado soportado por vhdl es el 2.147.483.647.

4.4.4 Comprobación lenguaje C

Al igual que para el algoritmo A se ha procedido a comprobar los resultados obtenidos de estas simulaciones de Modelsim mediante Turbo C++.

Se muestra el código para el cual se obtiene el resultado en 16 bits de salida al introducir cualquier número que contenga como máximo 32 bits de entrada.

```
#include <stdio.h>
/* Opcion C. Variables de 32 bits y 16 bits de salida. */
main()
{
 long unsigned x0, x1, y0, z0;
 int varcon, i, r1;
 do {
        printf("\nIntroducir un valor entero para x0: ");
        scanf("%lu", &x0);
        printf("\nIntroducir un valor entero para x1: ");
        scanf("%lu", &x1);
        printf("\nValor inicial de x0: %lu", x0);
        printf("\nValor inicial de x1: %lu", x1);
        x0= x0 + ((x0*x0) | 5);
        x1= x1 + ((x1*x1) | 13);
        printf("\nValor actual de x0: %lu", x0);
        printf("\nValor actual de x1: %lu", x1);
```

```
        //Non-linear function
        r1=32; /* Cambiando r1, se podria variar el numero de rondas */
        z0=x0;
        for (i=0; i<r1; i++) {
                z0=(z0<<1)+((z0+(0x56AB0A))>>1);
                y0=z0;
                y0=(y0>>1)+(y0<<1)+y0+x1;
                }
        z0=z0^y0;
        //Filter output
        //z0 es la salida del generador,
        //nos quedamos con los bits menos significativos que son los que varian
        //mas rapidamente.
        //para el ejemplo de variables de 16 bits, nos quedamos con los 8 bits
        //menos significativos.
        printf("\nValor decimal de z0: %lu", z0);
        printf("\nValor hexadecimal de z0: %lx", z0);
        z0=z0&0x0000ffff;
        printf("\nValor hexadecimal despues de la mascara de z0: %lx", z0);
        printf("\nValor decimal despues de la mascara de z0: %lu", z0);
        printf("\nPara continuar introducir 1, para parar introducir 0: ");
        scanf("%d", &varcon);
        } while (varcon != 0);
 printf("\nFin del programa");
 }
```

También se muestra el código en el cual se introducen 16 bits de entrada y se obtienen 8 bits de salida.

```
#include <stdio.h>
/* Opcion C. Variables de 16 bits y 8 bits de salida. */
main()
{
 unsigned x0, x1, y0, z0, varcon, i, r1;
 do {
        printf("\nIntroducir un valor entero para x0: ");
        scanf("%u", &x0);
        printf("\nIntroducir un valor entero para x1: ");
        scanf("%u", &x1);
        printf("\nValor inicial de x0: %u", x0);
        printf("\nValor inicial de x1: %u", x1);
        x0= x0 + ((x0*x0) | 5);
        x1= x1 + ((x1*x1) | 13);
        printf("\nValor actual de x0: %u", x0);
        printf("\nValor actual de x1: %u", x1);
        //Non-linear function
        r1=32; /* Cambiando r1, se podria variar el numero de rondas */
        z0=x0;
        for (i=0; i<r1; i++) {
                z0=(z0<<1)+((z0+(0x56AB0A))>>1);
                y0=z0;
                y0=(y0>>1)+(y0<<1)+y0+x1;
                }
  z0=z0^y0;
        //Filter output
        //z0 es la salida del generador,
        //nos quedamos con los bits menos significativos que son los que varian
        //mas rapidamente.
```

```
        //para el ejemplo de variables de 16 bits, nos quedamos con los 8 bits
        //menos significativos.
        printf("\nValor decimal de z0: %u", z0);
        printf("\nValor hexadecimal de z0: %x", z0);
        z0=z0&0x00ff;
        printf("\nValor hexadecimal despues de la mascara de z0: %x", z0);
        printf("\nValor decimal despues de la mascara de z0: %u", z0);
        printf("\nPara continuar introducir 1, para parar introducir 0: ");
        scanf("%d", &varcon);
        } while (varcon != 0);
printf("\nFin del programa");
}
```

La diferencia entre estos dos códigos radica en que, al igual que en el algoritmo A, para la opción de 32 bits de entrada hay que definir a los valores de x0, x1 y z0 como *long int* y no como entero simple como en la opción de 16 bits.

Se comprobó que el resultado de la simulación en Modelsim de la implementación de esta opción C de generador de números pseudo-aleatorios en vhdl concordaba con el resultado obtenido de la ejecución de este código tanto para 32 bits de entrada como para 16.

Al igual que en el algoritmo también se disponen de los ejecutables[9] para comprobar y obtener cuantos números se quieran con el generador de esta opción C. Para comprobar los resultados obtenidos de los ejecutables con el diseño en vhdl bastaría con cambiar los valores de las constantes seed_0 y seed_1 y calcular los de las constantes x0 y x1.

[9] Estos ejecutables se encuentran en el cd adjunto a este proyecto

4.4.5 Síntesis

4.4.5.1 Síntesis ISE

En este apartado se muestra la síntesis realizada de este diseño del algoritmo C por medio del programa ISE.

Al igual que en los dos algoritmos anteriores A y B, el objetivo es implementar este diseño en una FPGA lo más pequeña posible.

A su vez para los casos de menor número de bits a la entrada se ha vuelto a escoger una FPGA de la familia Spartan 3, más concretamente el dispositivo XC3S50, por su reducido y óptimo tamaño para nuestras especificaciones.

Property Name	Value
Product Category	General Purpose
Family	Spartan3
Device	XC3S50
Package	PQ208
Speed	-5

8 bits de entrada -> N = 4 bits de salida

Implementado con optimización en Área (Goal: Area, Effort: High)

Frecuencia máxima = 101,096 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	33	1,536	2%
Number of 4 input LUTs	87	1,536	5%
Logic Distribution			
Number of occupied Slices	47	768	6%
Number of Slices containing only related logic	47	47	100%
Number of Slices containing unrelated logic	0	47	0%
Total Number of 4 input LUTs	88	1,536	5%
Number used as logic	87		
Number used as a route-thru	1		
Number of bonded IOBs	8	124	6%
IOB Flip Flops	4		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal: Speed, Effort: High)

Frecuencia máxima = 150,64 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	33	1,536	2%
Number of 4 input LUTs	82	1,536	5%
Logic Distribution			
Number of occupied Slices	43	768	5%
Number of Slices containing only related logic	43	43	100%
Number of Slices containing unrelated logic	0	43	0%
Total Number of 4 input LUTs	82	1,536	5%
Number of bonded IOBs	8	124	6%
IOB Flip Flops	4		
Number of GCLKs	1	8	12%

16 bits de entrada -> N = 8 bits de salida

Implementado con optimización en Área

Frecuencia máxima = 115,496 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	57	1,536	3%
Number of 4 input LUTs	144	1,536	9%
Logic Distribution			
Number of occupied Slices	80	768	10%
Number of Slices containing only related logic	80	80	100%
Number of Slices containing unrelated logic	0	80	0%
Total Number of 4 input LUTs	156	1,536	10%
Number used as logic	144		
Number used as a route-thru	12		
Number of bonded IOBs	12	124	9%
IOB Flip Flops	8		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 118,06 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	57	1,536	3%
Number of 4 input LUTs	144	1,536	9%
Logic Distribution			
Number of occupied Slices	81	768	10%
Number of Slices containing only related logic	81	81	100%
Number of Slices containing unrelated logic	0	81	0%
Total Number of 4 input LUTs	156	1,536	10%
Number used as logic	144		
Number used as a route-thru	12		
Number of bonded IOBs	12	124	9%
IOB Flip Flops	8		
Number of GCLKs	1	8	12%

32 bits de entrada -> N = 16 bits de salida

Implementado en Área

Frecuencia máxima = 104,594 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	73	1,536	4%
Number of 4 input LUTs	267	1,536	17%
Logic Distribution			
Number of occupied Slices	155	768	20%
Number of Slices containing only related logic	155	155	100%
Number of Slices containing unrelated logic	0	155	0%
Total Number of 4 input LUTs	306	1,536	19%
Number used as logic	267		
Number used as a route-thru	39		
Number of bonded IOBs	20	124	16%
IOB Flip Flops	16		
Number of GCLKs	1	8	12%

Implementado en Velocidad

Frecuencia máxima = 106,055 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	73	1,536	4%
Number of 4 input LUTs	267	1,536	17%
Logic Distribution			
Number of occupied Slices	155	768	20%
Number of Slices containing only related logic	155	155	100%
Number of Slices containing unrelated logic	0	155	0%
Total Number of 4 input LUTs	306	1,536	19%
Number used as logic	267		
Number used as a route-thru	39		
Number of bonded IOBs	20	124	16%
IOB Flip Flops	16		
Number of GCLKs	1	8	12%

64 bits de entrada -> 48 bits de salida

Implementado con optimización en Área

Frecuencia máxima = 88,321 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	137	1,536	8%
Number of 4 input LUTs	437	1,536	28%
Logic Distribution			
Number of occupied Slices	279	768	36%
Number of Slices containing only related logic	279	279	100%
Number of Slices containing unrelated logic	0	279	0%
Total Number of 4 input LUTs	540	1,536	35%
Number used as logic	437		
Number used as a route-thru	103		
Number of bonded IOBs	36	124	29%
IOB Flip Flops	32		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 89,813 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	137	1,536	8%
Number of 4 input LUTs	451	1,536	29%
Logic Distribution			
Number of occupied Slices	279	768	36%
Number of Slices containing only related logic	279	279	100%
Number of Slices containing unrelated logic	0	279	0%
Total Number of 4 input LUTs	554	1,536	36%
Number used as logic	451		
Number used as a route-thru	103		
Number of bonded IOBs	36	124	29%
IOB Flip Flops	32		
Number of GCLKs	1	8	12%

128 bits de entrada -> N = 64 bits de salida

Implementado con optimización en Área

Frecuencia máxima = 67,23 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	265	1,536	17%
Number of 4 input LUTs	821	1,536	53%
Logic Distribution			
Number of occupied Slices	535	768	69%
Number of Slices containing only related logic	535	535	100%
Number of Slices containing unrelated logic	0	535	0%
Total Number of 4 input LUTs	1,052	1,536	68%
Number used as logic	821		
Number used as a route-thru	231		
Number of bonded IOBs	68	124	54%
IOB Flip Flops	64		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 68,091 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	265	1,536	17%
Number of 4 input LUTs	835	1,536	54%
Logic Distribution			
Number of occupied Slices	535	768	69%
Number of Slices containing only related logic	535	535	100%
Number of Slices containing unrelated logic	0	535	0%
Total Number of 4 input LUTs	1,066	1,536	69%
Number used as logic	835		
Number used as a route-thru	231		
Number of bonded IOBs	68	124	54%
IOB Flip Flops	64		
Number of GCLKs	1	8	12%

196 bits de entrada -> N = 98 bits de salida

Para esta implementación de 196 bits y para la de 256 bits de entrada a sido necesario la utilización de una FPGA más grande, y se ha optado por una de la familia Spartan 3 siendo el dispositivo XC3S200 el más pequeño en el que podíamos introducir nuestro diseño para cumplir con nuestros requerimientos de área.

Property Name	Value
Product Category	General Purpose
Family	Spartan3
Device	XC3S200
Package	PQ208
Speed	-5

Implementado con optimización en Área

Frecuencia máxima = 53,624 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	401	3,840	10%
Number of 4 input LUTs	1,229	3,840	32%
Logic Distribution			
Number of occupied Slices	807	1,920	42%
Number of Slices containing only related logic	807	807	100%
Number of Slices containing unrelated logic	0	807	0%
Total Number of 4 input LUTs	1,596	3,840	41%
Number used as logic	1,229		
Number used as a route-thru	367		
Number of bonded IOBs	102	141	72%
IOB Flip Flops	98		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 54,065 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	401	3,840	10%
Number of 4 input LUTs	1,243	3,840	32%
Logic Distribution			
Number of occupied Slices	807	1,920	42%
Number of Slices containing only related logic	807	807	100%
Number of Slices containing unrelated logic	0	807	0%
Total Number of 4 input LUTs	1,610	3,840	41%
Number used as logic	1,243		
Number used as a route-thru	367		
Number of bonded IOBs	102	141	72%
IOB Flip Flops	98		
Number of GCLKs	1	8	12%

256 bits de entrada -> N = 128 bits de salida

Implementado con optimización en Área

Frecuencia máxima = 45,499 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	521	3,840	13%
Number of 4 input LUTs	1,589	3,840	41%
Logic Distribution			
Number of occupied Slices	1,047	1,920	54%
Number of Slices containing only related logic	1,047	1,047	100%
Number of Slices containing unrelated logic	0	1,047	0%
Total Number of 4 input LUTs	2,076	3,840	54%
Number used as logic	1,589		
Number used as a route-thru	487		
Number of bonded IOBs	132	141	93%
IOB Flip Flops	128		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 45,816 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	521	3,840	13%
Number of 4 input LUTs	1,603	3,840	41%
Logic Distribution			
Number of occupied Slices	1,047	1,920	54%
Number of Slices containing only related logic	1,047	1,047	100%
Number of Slices containing unrelated logic	0	1,047	0%
Total Number of 4 input LUTs	2,090	3,840	54%
Number used as logic	1,603		
Number used as a route-thru	487		
Number of bonded IOBs	132	141	93%
IOB Flip Flops	128		
Number of GCLKs	1	8	12%

512 bits de entrada -> N = 256 bits de salida

En esta implementación de 512 bits de entrada se ha utilizado el dispositivo XC3S2000 siendo el más pequeño en el que podíamos implementar nuestro diseño.

Property Name	Value
Product Category	General Purpose
Family	Spartan3
Device	XC3S2000
Package	FG456
Speed	-5

Implementado con optimización en Área

Frecuencia máxima = 27,635 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,034	40,960	2%
Number of 4 input LUTs	3,127	40,960	7%
Logic Distribution			
Number of occupied Slices	2,075	20,480	10%
Number of Slices containing only related logic	2,075	2,075	100%
Number of Slices containing unrelated logic	0	2,075	0%
Total Number of 4 input LUTs	4,126	40,960	10%
Number used as logic	3,127		
Number used as a route-thru	999		
Number of bonded IOBs	260	333	78%
IOB Flip Flops	256		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 27,779 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,034	40,960	2%
Number of 4 input LUTs	3,141	40,960	7%
Logic Distribution			
Number of occupied Slices	2,076	20,480	10%
Number of Slices containing only related logic	2,076	2,076	100%
Number of Slices containing unrelated logic	0	2,076	0%
Total Number of 4 input LUTs	4,140	40,960	10%
Number used as logic	3,141		
Number used as a route-thru	999		
Number of bonded IOBs	260	333	78%
IOB Flip Flops	256		
Number of GCLKs	1	8	12%

1024 bits de entrada -> N = 512 bits de salida

Para esta implementación de 1024 bits de entrada se ha utilizado el dispositivo XC3S5000 siendo el más pequeño en el que podíamos implementar nuestro diseño.

Property Name	Value
Product Category	General Purpose
Family	Spartan3
Device	XC3S5000
Package	FG900
Speed	-5

Implementado con optimización en Área

Frecuencia máxima = 15,479 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	2,059	66,560	3%
Number of 4 input LUTs	6,208	66,560	9%
Logic Distribution			
Number of occupied Slices	4,129	33,280	12%
Number of Slices containing only related logic	4,129	4,129	100%
Number of Slices containing unrelated logic	0	4,129	0%
Total Number of 4 input LUTs	8,231	66,560	12%
Number used as logic	6,208		
Number used as a route-thru	2,023		
Number of bonded IOBs	516	633	81%
IOB Flip Flops	512		
Number of GCLKs	1	8	12%

Implementado con optimización en Tiempo (Goal Speed)

Frecuencia máxima = 15,525 MHz

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	2,059	66,560	3%
Number of 4 input LUTs	6,222	66,560	9%
Logic Distribution			
Number of occupied Slices	4,131	33,280	12%
Number of Slices containing only related logic	4,131	4,131	100%
Number of Slices containing unrelated logic	0	4,131	0%
Total Number of 4 input LUTs	8,245	66,560	12%
Number used as logic	6,222		
Number used as a route-thru	2,023		
Number of bonded IOBs	516	633	81%
IOB Flip Flops	512		
Number of GCLKs	1	8	12%

4.4.5.2 Resultados ISE

De esta síntesis realizada en el programa ISE para el algoritmo C se obtiene los siguientes resultados resumidos en la siguiente tabla en función de los parámetros que son de interés para este estudio (flips-flops, slices, tamaño área (LUTs) y frecuencia máxima).

Primeramente se muestran los datos obtenidos de esta síntesis con optimización en Área en la siguiente tabla.

	8 bits	16 bits	32 bits	64 bits	128 bits
Nº flip-flops	33	57	73	137	265
Nº slices	47	80	155	279	535
Nº LUTs	88	156	306	540	1052
Freq. Máx. (MHz)	101,096	115,496	104,594	88,321	67,23

	196 bits	256 bits	512 bits	1024 bits
Nº flip-flops	401	521	1034	2059
Nº slices	807	1047	2075	4129
Nº LUTs	1596	2076	4126	8231
Freq. Máx. (MHz)	53,624	45,499	27,635	15,479

En la siguiente gráfica resumen se muestran estos datos:

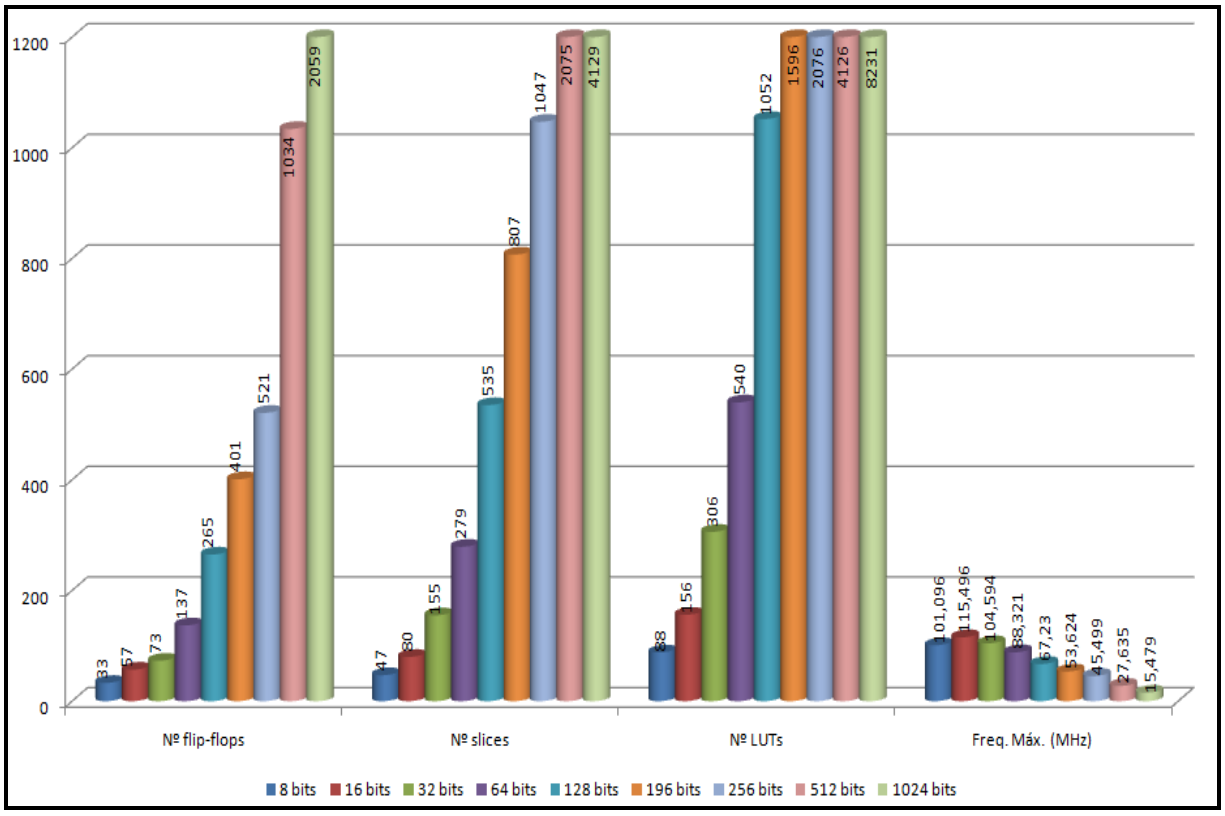

Figura 98- Gráfica resumen síntesis ISE con optimización en Área del algoritmo C.

La siguiente tabla resume los datos obtenidos de la síntesis con optimización en Tiempo.

	8 bits	16 bits	32 bits	64 bits	128 bits
Nº flip-flops	33	57	73	137	265
Nº slices	43	81	155	279	535
Nº LUTs	82	156	306	554	1066
Freq. Máx. (MHz)	150,64	118,06	106,055	89,813	68,091

	196 bits	256 bits	512 bits	1024 bits
Nº flip-flops	401	521	1034	2059
Nº slices	807	1047	2076	4131
Nº LUTs	1610	2090	4140	8245
Freq. Máx. (MHz)	54,065	45,816	27,779	15,525

Para ver reflejados estos datos se muestra la siguiente tabla:

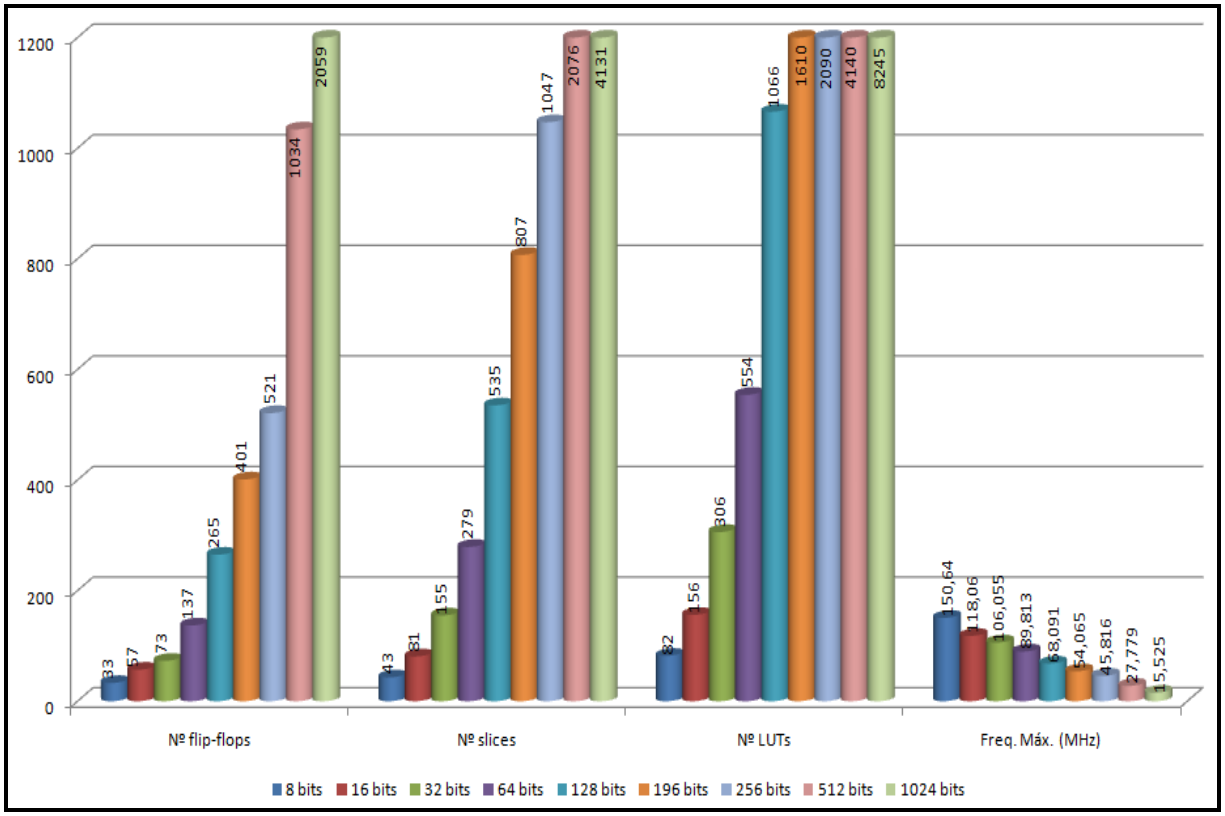

Figura 99- Gráfica resumen síntesis ISE con optimización en Tiempo del algoritmo C.

4.4.5.3 Síntesis SYNOPSYS

Se realiza la síntesis con la herramienta Design Vision del programa Synopsys en la que se ha usado la opción de un esfuerzo alto en área a la hora de

realizar esta síntesis, ya que para las especificaciones requeridas en este proyecto es necesario que se ocupe el menor área posible.

Para esta síntesis del algoritmo C se realiza también el estudio para los casos de 8, 16, 32, 64, 128, 196, 256, 512 y 1024 bits de entrada. En esta síntesis se extraen datos tanto de área como de consumo.

De los datos de área se obtendrá el número de *ports* que sería el número de puertos o entradas y salidas que se disponen en el diseño, el número de *nets* que sería el número de uniones o interconexionado, el número de *cells* que son el número total de las celdas o puertas lógicas usadas y el número de *references* que son los distintos tipos de puertas lógicas empleadas en la síntesis de este diseño. Por otra parte se dispondría del área perteneciente a la lógica combinacional y el área perteneciente a la lógica no combinacional además del *Net Interconnect area* que sería la parte de área ocupada por el interconexionado. Finalmente tendremos el *Total cell area* que sería el área total dispuesta para el cómputo total de las puertas lógicas, sumando la parte combinacional y la no combinacional; y si a este valor se le añade el área ocupada por el interconexionado se obtiene el *Total area.*

De los datos referidos al consumo se obtendrá la *Cell Internal Power* que se trata de la potencia consumida internamente por las celdas y la *Net Switching Power* que sería la potencia consumida asociada al interconexionado, que sumando estas dos potencias entre sí, se obtendría el consumo total dinámico para el diseño realizado ó *Total Dynamic Power*. A su vez también se obtendría el *Cell Leakage Power* que serían las pérdidas asociadas a este diseño.

8 bits de entrada -> N = 4 bits de salida

Datos de Área de la síntesis

```
Number of ports:                8
Number of nets:               226
Number of cells:              188
Number of references:          27

Combinational area:       1952.181973
Noncombinational area:     987.027988
Net Interconnect area:     124.311139

Total cell area:          2939.209961
Total area:               3063.521100
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  = 181.9102 uW   (67%)
  Net Switching Power  =  89.5644 uW   (33%)
                         ---------
Total Dynamic Power    = 271.4746 uW  (100%)

Cell Leakage Power     =  10.6589 uW
```

16 bits de entrada -> N = 8 bits de salida

Datos de Área de la síntesis

```
Number of ports:              12
Number of nets:              362
Number of cells:             318
Number of references:         32

Combinational area:       3272.892969
Noncombinational area:    1683.751972
Net Interconnect area:     249.699734

Total cell area:          4956.644941
Total area:               5206.344675
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  = 264.8226 uW   (68%)
  Net Switching Power  = 122.9987 uW   (32%)
                         ---------
Total Dynamic Power    = 387.8212 uW  (100%)

Cell Leakage Power     =  17.4962 uW
```

32 bits de entrada -> N = 16 bits de salida

Datos de Área de la síntesis

```
Number of ports:              20
Number of nets:              551
Number of cells:             445
Number of references:         33

Combinational area:       5984.902910
Noncombinational area:    2280.943958
Net Interconnect area:     423.875805

Total cell area:          8265.846868
```

```
Total area:               8689.722673
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  = 578.5972 uW   (55%)
  Net Switching Power  = 465.1396 uW   (45%)
                         ---------
Total Dynamic Power    =   1.0437 mW  (100%)

Cell Leakage Power     =  29.4618 uW
```

64 bits de entrada -> N = 48 bits de salida

Datos de Área de la síntesis

```
Number of ports:              36
Number of nets:             1028
Number of cells:             826
Number of references:         35

Combinational area:       11752.236805
Noncombinational area:    4271.583912
Net Interconnect area:     886.627381

Total cell area:          16023.820717
Total area:               16910.448098
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  = 865.4749 uW   (54%)
  Net Switching Power  = 746.8407 uW   (46%)
                         ---------
Total Dynamic Power    =   1.6123 mW  (100%)

Cell Leakage Power     =  57.9786 uW
```

128 bits de entrada -> N = 64 bits de salida

Datos de Área de la síntesis

```
Number of ports:              68
Number of nets:             2092
Number of cells:            1698
Number of references:         28

Combinational area:       23330.599669
Noncombinational area:    8252.863819
Net Interconnect area:    1953.426260
```

```
Total cell area:          31583.463488
Total area:               33536.889748
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   1.2265 mW   (66%)
  Net Switching Power  = 628.1184 uW   (34%)
                         ---------
Total Dynamic Power    =   1.8546 mW  (100%)

Cell Leakage Power     = 115.8783 uW
```

196 bits de entrada -> N = 98 bits de salida

Datos de Área de la síntesis

```
Number of ports:             102
Number of nets:             3186
Number of cells:            2588
Number of references:         25

Combinational area:       35553.820495
Noncombinational area:    12482.973721
Net Interconnect area:    3232.262680

Total cell area:          48036.794216
Total area:               51269.056895
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   1.5982 mW   (73%)
  Net Switching Power  = 595.3701 uW   (27%)
                         ---------
Total Dynamic Power    =   2.1936 mW  (100%)

Cell Leakage Power     = 175.0705 uW
```

256 bits de entrada -> N = 128 bits de salida

Datos de Área de la síntesis

```
Number of ports:             132
Number of nets:             4202
Number of cells:            3424
```

```
Number of references:          24

Combinational area:       46670.290353
Noncombinational area:    16215.423634
Net Interconnect area:    4128.142045

Total cell area:          62885.713987
Total area:               67013.856032
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   2.0410 mW   (73%)
  Net Switching Power  = 763.8121 uW   (27%)
                         ---------
Total Dynamic Power    =   2.8048 mW  (100%)

Cell Leakage Power     = 231.6215 uW
```

512 bits de entrada -> N = 256 bits de salida

Datos de Área de la síntesis

```
Number of ports:             260
Number of nets:             8467
Number of cells:            6921
Number of references:         24

Combinational area:       93780.731727
Noncombinational area:    32140.543262
Net Interconnect area:    9539.958073

Total cell area:          125921.274989
Total area:               135461.233062
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   3.9032 mW   (71%)
  Net Switching Power  =   1.5967 mW   (29%)
                         ---------
Total Dynamic Power    =   5.4999 mW  (100%)

Cell Leakage Power     = 470.1714 uW
```

1024 bits de entrada -> N = 512 bits de salida

Datos de Área de la síntesis

```
Number of ports:              516
Number of nets:             17107
Number of cells:            14024
Number of references:          26

Combinational area:       188606.988495
Noncombinational area:    63990.782520
Net Interconnect area:    21678.024587

Total cell area:          252597.771015
Total area:               274275.795603
```

Datos de Consumo de la síntesis

```
  Cell Internal Power  =   7.9325 mW   (71%)
  Net Switching Power  =   3.2461 mW   (29%)
                         ---------
Total Dynamic Power    =  11.1786 mW  (100%)

Cell Leakage Power     = 951.6962 uW
```

4.4.5.4 Resultados SYNOPSYS

En la siguiente tabla resumen se muestran todos los resultados obtenidos de esta síntesis a través de la herramienta Design Vision de Synopsys, en la que se muestran los tipos de datos obtenidos en función del número de bits del diseño:

	8 bits	16 bits	32 bits	64 bits	128 bits
Ports	8	12	20	36	68
Nets	226	362	551	1028	2092
Cells	188	318	445	826	1698
References	27	32	33	35	28
Comb. Area (µm²)	1.952	3.273	5.985	11.752	23.331
No comb. Area (µm²)	987	1.684	2.281	4.272	8.253
Net area (µm²)	124	250	424	887	1.953
Total cell area (µm²)	2.939	4.957	8.266	16.024	31.583
Total area (µm²)	3.064	5.206	8.690	16.910	33.537
Consumo total (mW)	0,27	0,39	1,04	1,61	1,85
Pérdidas (µW)	10,66	17,50	29,46	57,98	115,88

	196 bits	256 bits	512 bits	1024 bits
Ports	102	132	260	516
Nets	3186	4202	8467	17107
Cells	2588	3424	6921	14024
References	25	24	24	26
Comb. Area (µm²)	35.554	46.670	93.781	188.607
No comb. Area (µm²)	12.483	16.215	32.141	63.991
Net area (µm²)	3.232	4.128	9.540	21.678
Total cell area (µm²)	48.037	62.886	125.921	252.598
Total area (µm²)	51.269	67.014	135.461	274.276
Consumo total (mW)	2,19	2,80	5,50	11,18
Pérdidas (µW)	175,07	231,62	470,17	951,70

A continuación se muestra una gráfica con el número de puertas lógicas empleadas en cada diseño según los bits empleados a la entrada.

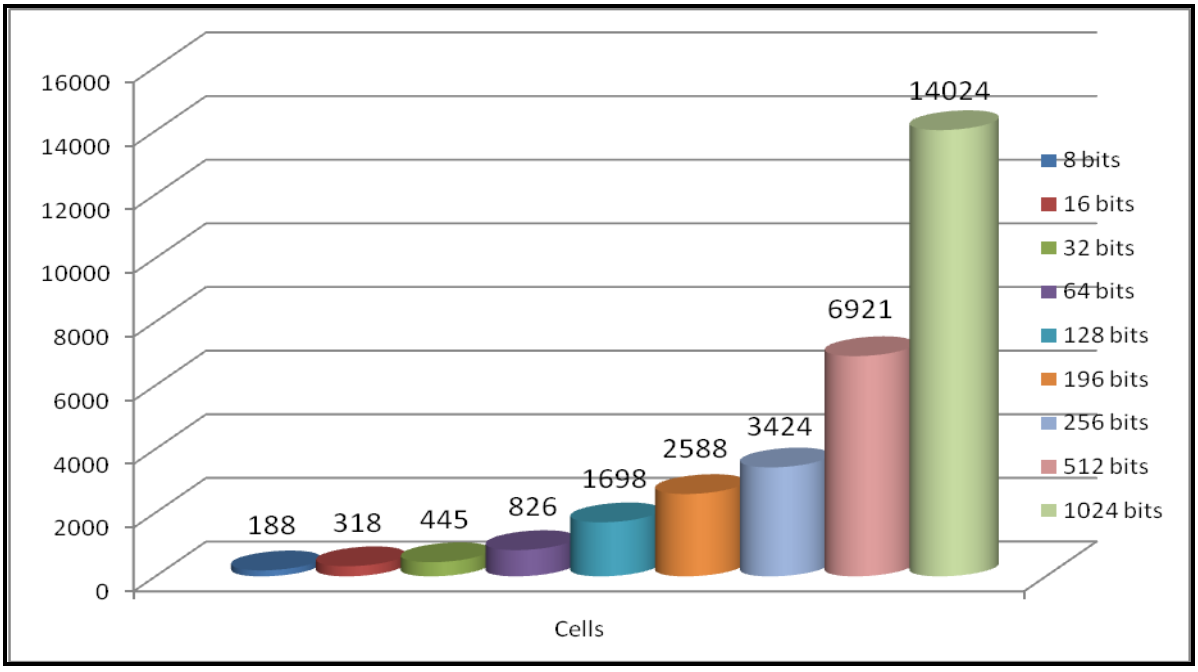

Figura 100.- Gráfica puertas lógicas del algortimo C.

En la siguiente gráfica se muestra el total del área según el número de bits empleados a la entrada con sus respectivas particiones en área correspondiente a la parte de la lógica combinacional y a la parte de la lógica no combinacional. Cabe destacar que para el total del área a parte de sumar estas dos partes correspondientes a la lógica, habría que añadirle la parte correspondiente al área del interconexionado.

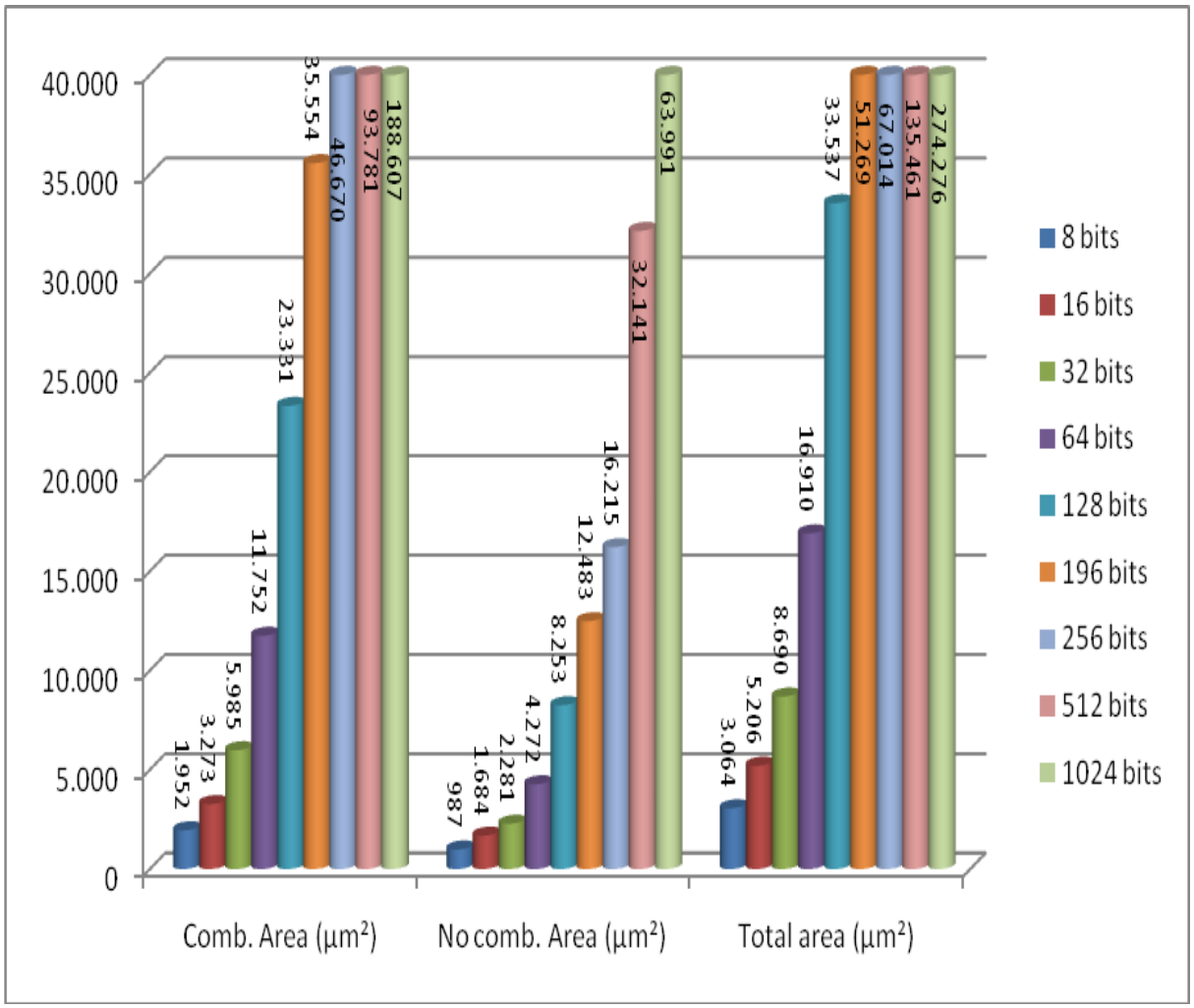

Figura 101.- Gráfica repartición área del algortimo C.

Se muestra la potencia que se consumiría (en unidades de miliwatios) por cada diseño en función del número de bits que se utilicen a la entrada por medio de la siguiente gráfica:

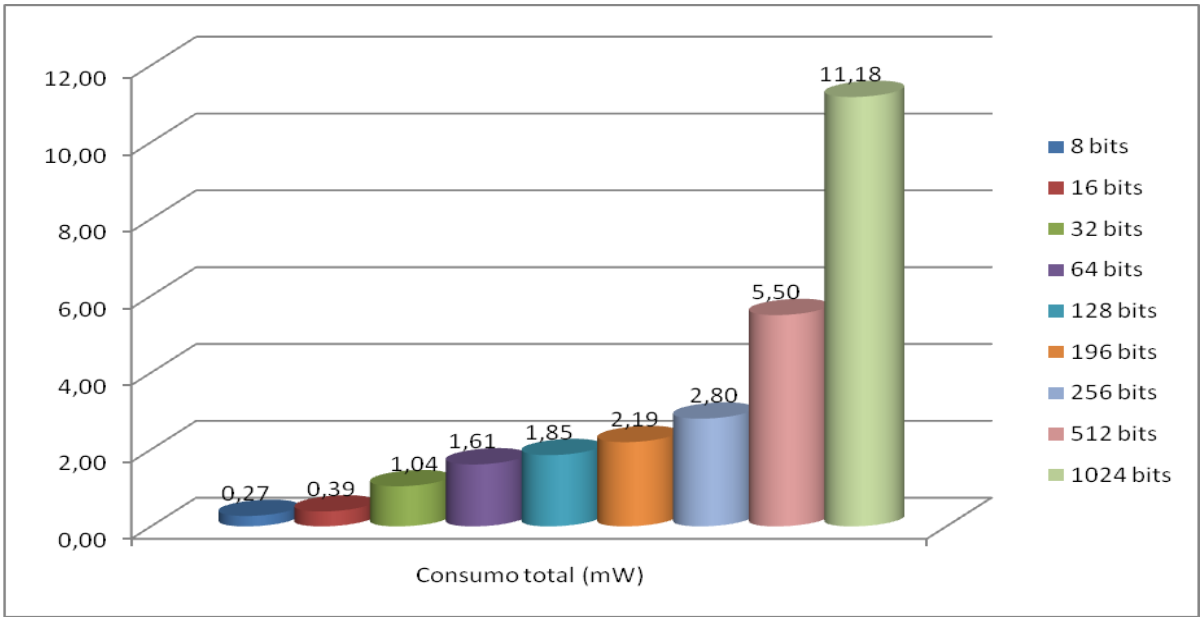

Figura 102.- Gráfica consumo del algortimo C.

4.4.6 Conclusiones

Una vez realizado el diseño de este generador de números pseudo-aleatorios para el algoritmo C en el lenguaje de descripción hardware vhdl y sus posteriores implementaciones tanto para la tecnología de dispositivos programables FPGAs como para la tecnología de circuitos a medida ASICs, se pueden analizar los resultados obtenidos y llegar a una serie de conclusiones a partir de los mismos.

Destacar nuevamente que mediante el ejecutable creado en Turbo C++, hemos comprobado que el diseño estaba realizado correctamente y posee un funcionamiento correcto de acuerdo a lo esperado.

De los resultados obtenidos en la implementación de dispositivos programables a partir del programa ISE, al igual que en el resto de los generadores anteriores se puede seguir observando que comparando las dos tablas de los dos tipos de optimizaciones llevadas a cabo, en el caso de la optimización efectuada con un alto esfuerzo en área se sigue consiguiendo reducir ligeramente el tamaño del área medida en LUTs, así como el número de flip-flops y el número de slices respecto a la optimización en tiempo; al igual, con la optimización en tiempo se sigue consiguiendo una frecuencia máxima ligeramente mayor aumentando la velocidad.

Asi mismo si para esta misma síntesis de dispositivos programables se compara ahora en función del número de bits a tratar en el proceso, se comprueba que también al igual que en los generadores anteriores a medida que aumenta el número de bits se va incrementando tanto el tamaño del área como el número de flip-flops y slices (figuras 98 y 99), también a cambio, se observa que a medida que se aumenta el número de bits a tratar se reduce sustancialmente la frecuencia máxima haciendo los procesos cada vez más lentos. Al igual que para el algoritmo A destacar nuevamente un dato que se sale de este patrón, es al igual el de la frecuencia máxima del caso de 8 bits para la optimización en área es menor que la de los casos de 16 y 32 bits siendo así más lento el proceso que en estos dos casos (se puede observar en la figura 98), esto será nuevamente debido a que al intentar optimizar en área lo máximo posible se reduzca la velocidad.

En lo referido a la síntesis realizada para la tecnología de las ASICs también se extrae la conclusión de que a medida que se aumenta el número de bits a tratar se incrementan las puertas lógicas, el interconexionado, el tamaño del área de estas celdas, ya sea en la parte correspondiente a la lógica combinacional como a la parte correspondiente a la lógica no combinacional.

Por último, nuevamente el estudio realizado del consumo de la potencia medida en miliwatios nos aporta que se consume bastante más a medida que aumenta el número de bits para estos generadores, lo mismo que para las pérdidas.

Conclusión del proyecto y líneas futuras

5 Conclusión del proyecto y líneas futuras

5.1 Conclusión del proyecto

Una vez llegada a la finalización de este proyecto es necesario pararse a reflexionar si se han alcanzado verdaderamente los objetivos de partida.

En primer lugar se ha realizado minuciosamente el estudio de los algoritmos que se han utilizado como generadores de números pseudo-aleatorios hasta que se ha llegado a su plena comprensión.

En segundo lugar se han diseñado por medio del lenguaje de descripción hardware vhdl estos algoritmos de los generadores, se han intentado realizar de la forma más genérica posible para dar lugar a los posibles análisis entre ellos. Para el caso de los algoritmos ultraligeros se ha comprobado con la ayuda de los ejecutables de Turbo C++ que esos diseños se habían realizado de forma satisfactoría.

Se ha realizado una primera implementación por medio de la tecnología basada en dispositivos programables en donde se han buscado FPGAs adecuadas para esa implementación y se han extraído los pertinentes parámetros referidos al área y al tiempo. En esta síntesis se ha realizado precisamente dos tipos de optimizaciones diferentes, una referida a un esfuerzo alto en área y la otra referida a otro esfuerzo alto pero en tiempo. Comparando estas optimizaciones, se ha observado como en el caso de la optimización en área se reducían ligeramente los parametrós referidos al tamaño y área (flip-flops, slices y LUTs), siendo este tipo de optimización muy recomendable para la tecnología RFID en la que se quieren aplicar estos generadores. Por el contrario, para la optimización en tiempo se observaba que se aumentaba la frecuencia máxima con lo que se disminuía el periodo mínimo y se ganaba en velocidad, aunque sin embargo esta faceta puede no resultar tan interesante para la tecnología RFID.

Igualmente se ha comprobado que a medida que se aumentaba el número de bits con el que se trabajaba se aumentaban los valores de los parámetros referidos al área, necesitando de un tamaño mayor para implementar, y disminuyendo la frecuencia máxima aumentando en lentitud el proceso. Con esto, para la tecnología RFID interesa por una parte que el generador sea de un número bajo de bits para ocupar el menor área posible (además de la rapidez), y por otra interesa un elevado número de bits en el generador para ganar en

seguridad para la encriptación, con lo que para elegir el más apropiado habrá que jugar con ello buscando un término medio.

Por otro lado, se ha realizado una segunda implementación para la tecnología de circuitos integrados a medida (implementación en las ASICs) en la que fue necesaria la dificultosa labor de la creación de las máquinas virtuales (con acceso a la red para la utilización de la licencia) para la utilización de la herramienta de ámbito profesional necesaria en este tipo de implementación, el Design Vision de Synopsys. Para facilitar esta labor en un futuro, en este proyecto se ha completado la creación de una máquina virtual con Linux Suse y Design Vision en correcto funcionamiento para poder ser utilizada esta herramienta directamente y con la facilidad añadida de poder ser utilizada en cualquier Pc, ya sea de la universidad o de ámbito privado, al ser completamente portable. Esta herramienta es muy útil de cara a implementaciones más rigurosas mientras que la herramienta utilizada en la primera parte de las implementaciones de este proyecto se asemejaba más a una aproximación.

Mediante la síntesis realizada por medio de esta herramienta para esta segunda tecnología se ha comprobado igualmente que a medida que se aumentaba el número de bits con el que se trabajaba se aumentaban considerablemente los parámetros referidos al tamaño (ya sea cualquier tipo de área como al número de celdas o puertas lógicas). A su vez también se ha realizado con esta herramienta el estudio del consumo y se ha observado que a medida que también aumentaba el número de bits aumentaba sustancialmente el consumo y lo mismo ocurría con las pérdidas asociadas a esta potencia consumida aunque en una magnitud mucho menor.

Finalmente, analizando los resultados obtenidos de cada uno de los diferentes tipos de generadores implementados se llega a una serie de conclusiones:

- El generador Blum Blum Shub es elevadamente superior en área a cualquiera de los tres PRNGs ultraligeros, debido en buena parte al multiplicador que usa para ello.
- Los algoritmos B y C son mayores en área que el algoritmo A puesto que en su diseño emplean un mayor número de operaciones e iteraciones en su máquina de estados, siendo el algoritmo B un poco mayor que el C.
- Por tanto el algoritmo A será el más pequeño de todos los generadores estudiados en lo que a área se refiere.

- Claramente el Blum Blum Shub es el generador que menor frecuencia máxima posee por lo que será el más lento.
- Sin embargo, las diferencias entre las frecuencias máximas de los tres PRNGs ultraligeros no son muy grandes, careciendo de tanta importancia.
- Respecto al consumo, el Blum Blum Shub tiene un consumo elevadísimo de la potencia, mientras que el algoritmo B y C lo tienen similar y al algoritmo A lo tiene algo menor. Al igual ocurre con las pérdidas asociadas a esta potencia consumida pero con un orden de magnitud bastante inferior.

5.2 Propuestas futuras

Para los posibles trabajos en el futuro se pueden considerar a tener en cuenta como partes a realizar las siguientes:

a) Búsqueda y realización de más generadores de números aleatorios, empleando nuevos algoritmos, los cuales se adapten al sistema que queremos utilizar, realizando comparaciones y estudios detallados, similares a los que se han realizado en este proyecto, tratando de que sea lo más apropiado posible para el caso de la tecnología RFID en la que se utilizará.

b) Realización de la implementación de un protocolo de encriptación completo, no sólo la adaptación de etiquetas (como en este caso en el que lo más importante es la generación de los números aleatorios) sino todo el diseño al completo, como por ejemplo el protocolo de encriptación SLAP.

Bibliografía

6 Bibliografía

[1] "RFID Labeling: Smart Labeling Concepts & Applications for the Consumer Packaged Goods Supply Chain (Paperback)", de Robert A. Kleist, Theodore A. Chapman, David A. Sakai y Brad S. Jarvis.

[2] "RFID Handbook: Fundamentals and Applications in Contactless Smart Cards and Identification", second edition (hardcover) de Finkenzeller, Klaus.

[3] Artículo "La RFID en el sector textil" de Jorge Moreno Cantón de Tag Ingenieros.

[4] "RFID: la tecnología de identificación por radiofrecuencia", de Muñoz Frías, José Daniel.

[5] Guía de seguridad de las TIC (CCN-STIC-401), Glosario y abreviaturas. Centro Criptológico Nacional.

[6] "A Simple Unpredictable Pseudo-Random Number Generator", *SIAM Journal on Computing*, de Lenore Blum, Manuel Blum, and Michael Shub.

[7] "Comparison of two pseudo-random number generators", *Advances in Cryptology: Proceedings of Crypto '82*, de Lenore Blum, Manuel Blum, and Michael Shub.

[8] "Vhdl. Lenguaje Para Síntesis Y Modelado De Circuitos". Editorial Rama, 2ª Edición Actualizada, de Pardo Carpio, Fernando.

[9] "Vhdl lenguaje estándar de diseño electrónico", McGraw Hill, Teres.

[10] "Manual básico de uso de la herramienta ModelSim", Departamento de Tecnología Electrónica de la escuela politécnica superior de la UC3M, de Casado Ortiz, Fernando y Mengibar Pozo, Luis.

[11] "Introducción al manejo de la herramienta de simulación ModelSim", de Sanz, César del dpto. de sistemas electrónicos y de control de la escuela universitaria de ingeniería técnica de telecomunicación de la UPM.

[12] "Xilinx ISE. Manual básico", Departamento de Tecnología Electrónica de la escuela politécnica superior de la UC3M, de Casado Ortiz, Fernando y Mengibar Pozo, Luis.

[13] "Guía de Introducción a Xilinx: Instalación, Diagrama Esquemático y Simulación", de Hans Rautenberg F.

[14] Nedjah, N. and Mourelle, L.M., Efficient hardware implementation of modular multiplication and exponentiation for public-key cryptography, Proceedings of the 5th. International Conference on High Performance Computing for Computational Science, Porto, Portugal, Lecture Notes in Computer Science, **2565**:451-463, Springer-Verlag, 2002.

[15] "Synopsys Unix Setup", Jinsik Yun, Jeannette Djigbenou, Dr. Dong S. Ha.

[16] "Synopsys – design compiler & design tools", Dpto. Arquitectura de Computadores y Automática de la UCM, de Mendías Cuadros, José Manuel.

[17] "Síntesis lógica usando Synopsys Design Compiler", de Mendías Cuadros, José Manuel.

[18] Design Compiler, User guide, Version B-2008.09, September 2008, Synopsys.

[19] Design Compiler, Tutorial using Design Vision, Version B-2008.09, September 2008, Synopsys.

[20] Design Vision, User guide, Version B-2008.09, September 2008, Synopsys.

[21] "Design Vision: A Logic Synthesis Tool", Jinsik Yun, Jeannette Djigbenou, Dr. Dong S.Ha.

[22] "Setup: Synopsys Design Vision", VLSI I Spring 2005.

[23] "Synopsys Design Vision", Digital System Design, Graduate Institute of Electronics Engineering, NTU, 25/03/2005.

[24] "Rápido tutorial de Synopsys Design Compiler", Ubuntu, electrónica y software libre, blog de Jara Berrocal, Abelardo.

[25] "Synopsys tutorial: Power estimation", Power estimation with Standard Cell Library and Synopsys tools, Bradley department of electrical & computer engineering by Syed Haider.

[26] "Synthesis Tools Installation Notes", Version B-2008.09, September 08, 2008, Synopsys.

[27] "Synopsys FPGA Licensing", User Guide, Version C-2009.03, March 2009, Synopsys.

[28] "¿Cómo configurar IP en Sun Solaris?", por Ponce López, Oscar.

[29] "Guía de instalación de Solaris 10: instalaciones básicas", Sun Microsystems.

[30] "Network guide", Sun Microsystems.

[31] "Guía de instalación de Solaris 10: instalaciones basadas en red", Sun Microsystems.

[32] "Applied Cryptography Second Edition: Protocols, algorithms, and source code in C". John Wiley & Sons, Inc., 1996 by Schneier, Bruce.

ANEXO A

Códigos vhdl empleados

Anexo A - Códigos vhdl empleados

A.1 Código Blum Blum Shub

Código empleado para un nível de seguridad N=128 bits

```
library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_unsigned.all;
    use ieee.std_logic_arith.all;


--ENTIDAD


    ENTITY bbs IS

        GENERIC(N:INTEGER:=128);
        PORT(
            Semilla: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            M: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Salida: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Inicio: IN STD_LOGIC;
            Fin: OUT STD_LOGIC;
            CLK,RESET: IN STD_LOGIC;
            Fin_termino_i: OUT STD_LOGIC
            );
        END bbs;


--ARQUITECTURA


  ARCHITECTURE bbs_a OF bbs IS

      -- Componente

      COMPONENT mont_CARD
        GENERIC(N:INTEGER:=128);
        PORT(
            A,B: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            M: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            S: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            INICIO: IN STD_LOGIC;
            FIN: OUT STD_LOGIC;
            CLK,RESET: IN STD_LOGIC
            );
        END COMPONENT;


      --Señales
```

```
      TYPE ESTADO IS
(reposo,Activar_montgomery,Calculo_montgomery,Fin_calculo_termino,Fin_calcu
lo,Estado_final);
    SIGNAL ACTUAL,SIGUIENTE:ESTADO;
    SIGNAL
Aux_A_montgomery,Aux_B_montgomery,Salida_montgomery:STD_LOGIC_VECTOR(N-1
DOWNTO 0);
    SIGNAL CONTADOR,N_terminos_sucesion: INTEGER RANGE 0 TO N;
    SIGNAL Fin_montgomery:STD_LOGIC;
    SIGNAL Inicio_montgomery: STD_LOGIC;


      BEGIN

      --INSTANCIACIÓN

      MI_BLUM: MONT_CARD PORT MAP
(A=>Aux_A_montgomery,B=>Aux_B_montgomery,INICIO=>Inicio_montgomery,CLK=>CLK
,RESET=>RESET,FIN=>Fin_montgomery,M=>M,S=>Salida_montgomery);



       -- PROCESO PARA LA MAQUINA DE ESTADOS

        PROCESS(CLK,RESET)
                    BEGIN

                        IF RESET='1' THEN
                            actual<=reposo;

                         elsif clk 'EVENT AND CLK='1' THEN
                             ACTUAL<=SIGUIENTE;

                         END IF;
                      END PROCESS;


       --MAQUINA DE ESTADOS

          PROCESS
(ACTUAL,Fin_montgomery,CONTADOR,Inicio,N_terminos_sucesion,Semilla)
              BEGIN

                  CASE ACTUAL IS

                      WHEN REPOSO=>
                          Fin_termino_i<='0';
                          Fin<='0';
                          Inicio_montgomery<='0';
                          IF Inicio='1' THEN
                              SIGUIENTE<=Activar_montgomery;
                              ELSE
                                SIGUIENTE<=ACTUAL;
                          END IF;

                        WHEN Activar_montgomery=> --Se activa EL BIT
Inicio_montgomery para hacer montgomery
                            Fin_termino_i<='0';
                            Fin<='0';
```

```
                            Inicio_montgomery<='1';
                            SIGUIENTE<=Calculo_montgomery;

                        WHEN Calculo_montgomery=>
                            Fin_termino_i<='0';
                            Fin<='0';
                            Inicio_montgomery<='1';
                            IF Fin_montgomery='1' THEN

                                --si CONTADOR = N_terminos_sucesion
finaliza el calculo por montgomery
                                --sino continua con el calculo de los
siguientes terminos de la sucesion

                              IF CONTADOR=N_terminos_sucesion THEN
                                   SIGUIENTE<=Fin_calculo;
                                   ELSE SIGUIENTE<=Fin_calculo_termino;
                               END IF;
                            ELSE
                               SIGUIENTE<=ACTUAL;
                           END IF;


                        WHEN Fin_calculo_termino=>
                            Fin_termino_i<='1';
                            Fin<='0';
                            Inicio_montgomery<='0';
                            SIGUIENTE<=Calculo_montgomery;


                        WHEN Fin_calculo=>
                            Fin_termino_i<='1';
                            Fin<='0';
                            Inicio_montgomery<='0';
                            SIGUIENTE<=Estado_final;

                        WHEN Estado_final=> --FIN DEL ALGORITMO
                            Fin_termino_i<='0';
                            Inicio_montgomery<='0';
                            Fin<='1';
                            SIGUIENTE<=REPOSO;
                        END CASE;
                    END PROCESS;



 -------------- PROCESO SECUENCIAL


        PROCESS(CLK,RESET)
                    BEGIN

                         IF RESET='1' THEN
                             CONTADOR<=0;
                             N_terminos_sucesion<=0;
                             Aux_A_montgomery<=Semilla;
                             Aux_B_montgomery<=Semilla;

                          elsif clk 'EVENT AND CLK='1' THEN
```

```
                                IF ACTUAL=REPOSO THEN
                                    CONTADOR<=0;

                                  -- ESTE REGISTRO INDICA LA CANTIDAD DE
NUMEROS ALEATORIOS QUE SE QUIERE CALCULAR
                                    N_terminos_sucesion<=5;

                                    ELSIF ACTUAL=Activar_montgomery THEN
                                        Aux_A_montgomery<=Semilla;
                                        Aux_B_montgomery<=Semilla;


                                    ELSIF ACTUAL=Fin_calculo_termino THEN
                                         Aux_A_montgomery<=Salida_montgomery;
                                         Aux_B_montgomery<=Salida_montgomery;
                                         CONTADOR<=CONTADOR+1;
                                         Salida<=Salida_montgomery;
                                    ELSIF ACTUAL=Fin_calculo THEN
                                        Salida<=Salida_montgomery;
                                    ELSIF ACTUAL=Estado_final THEN

                                    END IF;

                            END IF;
                        END PROCESS;

                    END bbs_a;
```

Código empleado del multiplicador Montgomery como componente

```
library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_unsigned.all;
    use ieee.std_logic_arith.all;

    --ENTIDAD
    entity mont_CARD IS
        GENERIC(N:INTEGER:=128);
        PORT(
            A,B: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            M: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            S: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            INICIO: IN STD_LOGIC;
            FIN: OUT STD_LOGIC;
            CLK,RESET: IN STD_LOGIC
            );
                END mont_CARD;


    architecture mont_CARD_a of mont_CARD is
    TYPE ESTADO IS (reposo,estado1,estado2,ESTADO3,FINAL);

    --DECLARACION DE SEÑALES
    SIGNAL ACTUAL,SIGUIENTE:ESTADO;
```

```
    SIGNAL R1:STD_LOGIC_VECTOR(2*N-1 DOWNTO 0);
    SIGNAL CONTADOR: INTEGER RANGE 0 TO N-1;
    SIGNAL CC,DD: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
    SIGNAL E: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
    BEGIN


    --PROCESO SECUENCIAL DE LOS ESTADOS
        PROCESS(CLK,RESET)
                    BEGIN

                         IF RESET='1' THEN

                             actual<=reposo;

                          elsif clk 'EVENT AND CLK='1' THEN

                              ACTUAL<=SIGUIENTE;
                              if contador=n-1 OR ACTUAL=REPOSO OR
ACTUAL=ESTADO2 then
                              contador<=0;
                              else contador<=contador+1;
                          end if;
                          END IF;
                      END PROCESS;



        --PROCESO COMBINACIONAL

        PROCESS (CONTADOR,ACTUAL,INICIO)

            BEGIN

                CASE ACTUAL IS

                    WHEN REPOSO=>
                        FIN<='0';

                        IF INICIO='1' THEN
                        SIGUIENTE<=ESTADO1;
                        ELSE SIGUIENTE<=ACTUAL;
                    END IF;


                    WHEN ESTADO1=> --PARTE DEL ALGORITMO QUE SE HACE
(A,B,M)
                        FIN<='0';

                        IF CONTADOR=N-1 THEN

                        SIGUIENTE<=ESTADO2;

                        ELSE

                        SIGUIENTE<=ACTUAL;
                        END IF;

                    WHEN ESTADO2=> --SE HACE OTRA VEZ EL ALGORITMO CON
(R,E,M)
```

```
                        FIN<='0';
                        SIGUIENTE<=ESTADO3;


                    WHEN ESTADO3=>
                         FIN<='0';
                         IF CONTADOR=N-1 THEN
                          SIGUIENTE<=FINAL;
                          ELSE SIGUIENTE<=ACTUAL;
                      END IF;


                    WHEN FINAL=>
                        FIN<='1';
                        SIGUIENTE<=REPOSO;


                    END CASE;
                END PROCESS;



               --PROCESO SECUENCIAL

                PROCESS(CLK,RESET)
                    VARIABLE C,D:STD_LOGIC_VECTOR(N-1 DOWNTO 0);
                    VARIABLE R:STD_LOGIC_VECTOR(2*N-1 DOWNTO 0);
                    VARIABLE Q:STD_LOGIC_VECTOR(N-1 DOWNTO 0);

                    BEGIN

                         IF RESET='1' THEN
                           R:=CONV_STD_LOGIC_VECTOR(0,2*N);
                           E<=CONV_STD_LOGIC_VECTOR(79970,N);   --ESTA
SEÑAL ES LA CONSTANTE QUE VALE 4^N MOD M
                           C:=CONV_STD_LOGIC_VECTOR(0,N);
                           r1<=conv_std_logic_vector(0,2*n);
                           CC<=CONV_STD_LOGIC_VECTOR(0,N);
                           DD<=CONV_STD_LOGIC_VECTOR(0,N);
                           D:=CONV_STD_LOGIC_VECTOR(0,N);

                          elsif clk 'EVENT AND CLK='1' THEN

                              IF ACTUAL=REPOSO THEN
                                  r1<=conv_std_logic_vector(0,2*n);
                                  CC<=A;                                --
SE CARGA EN CC Y DD LOS DATOS DE LAS ENTRADAS
                                  DD<=B;
                                  Q:=CONV_STD_LOGIC_VECTOR(0,N);
                                  R:=CONV_STD_LOGIC_VECTOR(0,2*N);
                                  S<=CONV_STD_LOGIC_VECTOR(0,N);


                                ELSIF ACTUAL=ESTADO1 OR ACTUAL=ESTADO3 THEN


                                      IF CC(CONTADOR)='1' THEN R:=R+DD;
                                      ELSE C:=CONV_STD_LOGIC_VECTOR(0,N);
                                      END IF;
```

```
                                        IF R(0)='0' THEN R:='0' & R(2*N-1
DOWNTO 1);
                                        Else
                                        R:=R+M;
                                        R:='0' & R(2*N-1 DOWNTO 1);
                                        END IF;


                                    ELSIF ACTUAL=ESTADO2 THEN
                                     -- EN ESTE ESTADO SE REALIZA DE NUEVO
EL ALGORITMO DE MONTGOMERY, SE CARGA EN D LA CONSTANTE E

                                         CC<=R1(n-1 downto 0);
                                         DD<=E;
                                         R1<=CONV_STD_LOGIC_VECTOR(0,2*N);
                                         R:=CONV_STD_LOGIC_VECTOR(0,2*N);

                                    END IF;

                                  R1<=R;
                                  S<=R(N-1 DOWNTO 0);


                            END IF;
                        END PROCESS;

            END MONT_CARD_A;
```

A.2 Códigos algoritmo A

Código empleado para N = 16 bits de salida (32 bits de entrada)

```
library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_unsigned.all;
    use ieee.std_logic_arith.all;


--ENTIDAD

    ENTITY prngA IS
        GENERIC(N:INTEGER:=16);
        PORT(

            z0: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);

           INICIO: IN STD_LOGIC;
            FIN: OUT STD_LOGIC;
            CLK,RESET: IN STD_LOGIC
            );
    END prngA;



--ARQUITECTURA

    architecture opcionA of prngA is



--DECLARACION DE SEÑALES
    TYPE ESTADO IS (Reposo,Iterando,Filtro_salida);
    SIGNAL ACTUAL,SIGUIENTE:ESTADO;

    CONSTANT seed_0: STD_LOGIC_VECTOR((2*N-1) DOWNTO 0):=
CONV_STD_LOGIC_VECTOR (55555555,(2*N));
    CONSTANT seed_1: STD_LOGIC_VECTOR((2*N-1) DOWNTO 0):=
CONV_STD_LOGIC_VECTOR (777777777,(2*N));

    CONSTANT x0: STD_LOGIC_VECTOR((2*N-1) DOWNTO 0):= CONV_STD_LOGIC_VECTOR
(413433136,(2*N));    -- x0=x0+((x0*x0)|5)
    CONSTANT x1: STD_LOGIC_VECTOR((2*N-1) DOWNTO 0):= CONV_STD_LOGIC_VECTOR
(206293086,(2*N));     -- x1=x1+((x1*x1)|13)


    SIGNAL z: STD_LOGIC_VECTOR((2*N-1) DOWNTO 0);
    SIGNAL contador_i: INTEGER RANGE 0 TO 64;

    BEGIN
```

```
       -- PROCESO PARA LA MAQUINA DE ESTADOS

        PROCESS(CLK,RESET)
                    BEGIN

                         IF reset='1' THEN
                             actual<=reposo;

                          elsif clk 'EVENT AND clk='1' THEN
                              actual<=siguiente;

                         END IF;
                       END PROCESS;


        --MAQUINA DE ESTADOS

          PROCESS (ACTUAL,Inicio,contador_i)
              BEGIN

                   CASE ACTUAL IS

                         WHEN Reposo=>

                           IF Inicio='1' THEN
                               SIGUIENTE<=Iterando;
                            ELSE
                               SIGUIENTE<=ACTUAL;
                           END IF;



                         WHEN Iterando=>
                            IF contador_i=64 THEN
                               SIGUIENTE<=Filtro_salida;
                             ELSE

                                SIGUIENTE<=ACTUAL;
                            END IF;

                         WHEN Filtro_salida=>
                             fin<='1';
                             SIGUIENTE<=Reposo;
                         END CASE;
                    END PROCESS;


  --PROCESO SECUENCIAL PARA EL CONTADOR DE LAS 64 ITERACIONES

      PROCESS(reset,clk)
            BEGIN

            IF reset='1' THEN

                       contador_i <= 0;

              elsif clk'EVENT AND CLK='1' THEN
                IF ACTUAL=Reposo THEN
                    contador_i <= 0;
```

```
                    z <= x0;


                ELSIF ACTUAL=Iterando THEN
                     IF  contador_i<64 THEN
                       z <= ('0' & z(31 downto 1)) + (z(30 downto 0) & '0')
+ z + x1;  --desplazamiento;relleno con ceros
                        contador_i <= contador_i + 1;
                      ELSE
                        contador_i <= 0;
                     END IF;
                ELSIF ACTUAL=Filtro_salida THEN
                        z0 <= z(15 downto 0);  --me quedo con los 16 bits
menos significativos

                END IF;
             END IF;
      END PROCESS;


  END opcionA;
```

A.3 Códigos algoritmo B

Código empleado para N = 16 bits de salida (32 bits de entrada)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


--ENTIDAD

ENTITY prngB IS
GENERIC(N:INTEGER:=16);
PORT(

z0: OUT unsigned(N-1 DOWNTO 0);

INICIO: IN STD_LOGIC;
FIN: OUT STD_LOGIC;
CLK,RESET: IN STD_LOGIC
);
END prngB;



--ARQUITECTURA

architecture opcionb of prngB is



--DECLARACION DE SEÑALES
TYPE ESTADO IS
(Reposo,IterandoZ,Establecer_y0,IterandoY,Calculo_Z_final,Filtro_salida);
SIGNAL ACTUAL,SIGUIENTE:ESTADO;

CONSTANT seed_0: unsigned((2*N-1) DOWNTO 0):= x"034FB5E3"; --55.555.555
CONSTANT seed_1: unsigned((2*N-1) DOWNTO 0):= x"2E5BF271"; --777.777.777

CONSTANT x0: unsigned((2*N-1) DOWNTO 0):= x"18A47D30";   --
x0=x0+((x0*x0)|5)
CONSTANT x1: unsigned((2*N-1) DOWNTO 0):= x"0C4BC85E";   --
x1=x1+((x1*x1)|13)

SIGNAL z,y0: unsigned((2*N-1) DOWNTO 0);
SIGNAL contador_z,contador_y: INTEGER RANGE 0 TO 24;

BEGIN



-- PROCESO PARA LA MAQUINA DE ESTADOS

PROCESS(CLK,RESET)
```

```
BEGIN

IF reset='1' THEN
actual<=reposo;

elsif clk 'EVENT AND clk='1' THEN
actual<=siguiente;

END IF;
END PROCESS;


--MAQUINA DE ESTADOS-------------------------------------------------------


PROCESS (ACTUAL,Inicio,contador_z,contador_y)
BEGIN

CASE ACTUAL IS

WHEN Reposo=>

IF Inicio='1' THEN
SIGUIENTE<=IterandoZ;
ELSE
SIGUIENTE<=ACTUAL;
END IF;



WHEN IterandoZ=>
IF contador_z=24 THEN
SIGUIENTE<=Establecer_y0;
ELSE

SIGUIENTE<=ACTUAL;
END IF;

WHEN Establecer_y0=>


SIGUIENTE<=IterandoY;


WHEN IterandoY=>
IF contador_y=24 THEN
SIGUIENTE<=Calculo_Z_final;
ELSE

SIGUIENTE<=ACTUAL;
END IF;

WHEN Calculo_Z_final=>


SIGUIENTE<=Filtro_salida;


WHEN Filtro_salida=>
fin<='1';
```

```
SIGUIENTE<=Reposo;
END CASE;
END PROCESS;


--PROCESO SECUENCIAL-------------------------------------------------------


PROCESS(reset,clk)
BEGIN

IF reset='1' THEN

contador_z <= 0;
contador_y <= 0;

elsif clk'EVENT AND CLK='1' THEN
IF ACTUAL=Reposo THEN
contador_z <= 0;
contador_y <= 0;

z <= x0 XOR x1;

ELSIF ACTUAL=IterandoZ THEN
IF  contador_z<24 THEN
z <= shift_left(z,1) + shift_right((z + x"56AB0A"),1);
contador_z <= contador_z + 1;
ELSE
contador_z <= 0;
END IF;
ELSIF ACTUAL=Establecer_y0 THEN
y0 <= x1 XOR z;
ELSIF ACTUAL=IterandoY THEN
IF  contador_y<24 THEN
y0 <= shift_right(y0,1) + shift_left(y0,1) + y0 + x"72A4FB";
contador_y <= contador_y + 1;
ELSE
contador_y <= 0;
END IF;
ELSIF ACTUAL=Calculo_Z_final THEN
z <= z XOR y0;

ELSIF ACTUAL=Filtro_salida THEN
z0 <= z(15 downto 0);  --me quedo con los 16 bits menos significativos

END IF;
END IF;
END PROCESS;


END opcionb;
```

Código empleado para N = 8 bits de salida (16 bits de entrada)

ejemplo del caso para el que surgen problemas en la simulación

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


--ENTIDAD

ENTITY prngB IS
GENERIC(N:INTEGER:=8);
PORT(

z0: OUT unsigned(N-1 DOWNTO 0);

INICIO: IN STD_LOGIC;
FIN: OUT STD_LOGIC;
CLK,RESET: IN STD_LOGIC
);
END prngB;



--ARQUITECTURA

architecture opcionb of prngB is



--DECLARACION DE SEÑALES
TYPE ESTADO IS
(Reposo,IterandoZ,AuxliarZ,Establecer_y0,IterandoY,AuxliarY,Calculo_Z_final
,Filtro_salida);
SIGNAL ACTUAL,SIGUIENTE:ESTADO;

CONSTANT seed_0: unsigned((2*N-1) DOWNTO 0):= x"15B3"; --5.555(n°s
decimales a usar
                                      --como semilla para 16 bits
                                      --de entrada)
CONSTANT seed_1: unsigned((2*N-1) DOWNTO 0):= x"1E61"; --7.777

CONSTANT x0: unsigned((2*N-1) DOWNTO 0):= x"F0E0";   -- x0=x0+((x0*x0)|5)
CONSTANT x1: unsigned((2*N-1) DOWNTO 0):= x"FF2E";   -- x1=x1+((x1*x1)|13)


SIGNAL z,y0: unsigned((2*N-1) DOWNTO 0);
SIGNAL z_AUX,y0_AUX: unsigned(23 DOWNTO 0); --Señal auxiliar para poder
emplear los
                         --n°s hexadecinales al ser de dimensiones
                         --mayores a la señal a tratar cuya dimensión es
                         --el n° de bits a la entrada
SIGNAL contador_z,contador_y: INTEGER RANGE 0 TO 24;

BEGIN
```

```
-- PROCESO PARA LA MAQUINA DE ESTADOS

PROCESS(CLK,RESET)
BEGIN

IF reset='1' THEN
actual<=reposo;

elsif clk 'EVENT AND clk='1' THEN
actual<=siguiente;

END IF;
END PROCESS;


--MAQUINA DE ESTADOS-------------------------------------------------------


PROCESS (ACTUAL,Inicio,contador_z,contador_y)
BEGIN

CASE ACTUAL IS

WHEN Reposo=>

IF Inicio='1' THEN
SIGUIENTE<=IterandoZ;
ELSE
SIGUIENTE<=ACTUAL;
END IF;



WHEN IterandoZ=>
IF contador_z=24 THEN
SIGUIENTE<=Establecer_y0;
ELSE
SIGUIENTE<=AuxliarZ;
END IF;

WHEN AuxliarZ=>

SIGUIENTE<=IterandoZ;


WHEN Establecer_y0=>

SIGUIENTE<=IterandoY;


WHEN IterandoY=>
IF contador_y=24 THEN
SIGUIENTE<=Calculo_Z_final;
ELSE
SIGUIENTE<=AuxliarY;
END IF;

WHEN AuxliarY=>
```

```
SIGUIENTE<=IterandoY;


WHEN Calculo_Z_final=>

SIGUIENTE<=Filtro_salida;


WHEN Filtro_salida=>
fin<='1';
SIGUIENTE<=Reposo;
END CASE;
END PROCESS;


--PROCESO SECUENCIAL-----------------------------------------------------


PROCESS(reset,clk)
BEGIN

IF reset='1' THEN

contador_z <= 0;
contador_y <= 0;

elsif clk'EVENT AND CLK='1' THEN
IF ACTUAL=Reposo THEN
contador_z <= 0;
contador_y <= 0;

z <= x0 XOR x1;

ELSIF ACTUAL=IterandoZ THEN
IF  contador_z<24 THEN
z_AUX <= shift_left(z,1) + shift_right((z + x"56AB0A"),1);
contador_z <= contador_z + 1;
ELSE
contador_z <= 0;
END IF;
ELSIF ACTUAL=AuxliarZ THEN
   z <= z_AUX(2*N-1 downto 0);

ELSIF ACTUAL=Establecer_y0 THEN
y0 <= x1 XOR z;
ELSIF ACTUAL=IterandoY THEN
IF  contador_y<24 THEN
y0_AUX <= shift_right(y0,1) + shift_left(y0,1) + y0 + x"72A4FB";
contador_y <= contador_y + 1;
ELSE
contador_y <= 0;
END IF;

ELSIF ACTUAL=AuxliarY THEN
   y0 <= y0_AUX(2*N-1 downto 0);

ELSIF ACTUAL=Calculo_Z_final THEN
z <= z XOR y0;

ELSIF ACTUAL=Filtro_salida THEN
```

```
z0 <= z(N-1 downto 0);  --me quedo con los 8 bits menos significativos

END IF;
END IF;
END PROCESS;


END opcionb;
```

A.4 Códigos algoritmo C

Código empleado para N = 16 bits de salida (32 bits de entrada)

```
library ieee;
    use ieee.std_logic_1164.all;
    use ieee.numeric_std.all;


--ENTIDAD

    ENTITY prngC IS
        GENERIC(N:INTEGER:=16);
        PORT(

            z0: OUT unsigned(N-1 DOWNTO 0);

           INICIO: IN STD_LOGIC;
            FIN: OUT STD_LOGIC;
            CLK,RESET: IN STD_LOGIC
            );
    END prngC;



--ARQUITECTURA

    architecture opcionC of prngC is



--DECLARACION DE SEÑALES
    TYPE ESTADO IS
(Reposo,IterandoZ,PasoZaY0,IterandoY,Calculo_Z_final,Filtro_salida);
    SIGNAL ACTUAL,SIGUIENTE:ESTADO;

    CONSTANT seed_0: unsigned((2*N-1) DOWNTO 0):= x"034FB5E3"; --55.555.555
    CONSTANT seed_1: unsigned((2*N-1) DOWNTO 0):= x"2E5BF271"; --
777.777.777

    CONSTANT x0: unsigned((2*N-1) DOWNTO 0):= x"18A47D30";   --
x0=x0+((x0*x0)|5)
    CONSTANT x1: unsigned((2*N-1) DOWNTO 0):= x"0C4BC85E";   --
x1=x1+((x1*x1)|13)

    SIGNAL z,y0: unsigned((2*N-1) DOWNTO 0);
    SIGNAL contador_i: INTEGER RANGE 0 TO 32;
    BEGIN




       -- PROCESO PARA LA MAQUINA DE ESTADOS
```

```
PROCESS(CLK,RESET)
            BEGIN

                 IF reset='1' THEN
                     actual<=reposo;

                  elsif clk 'EVENT AND clk='1' THEN
                      actual<=siguiente;

                  END IF;
              END PROCESS;


--MAQUINA DE ESTADOS-----------------------------------------------


  PROCESS (ACTUAL,Inicio,contador_i)
      BEGIN

          CASE ACTUAL IS

               WHEN Reposo=>

                 IF Inicio='1' THEN
                     SIGUIENTE<=IterandoZ;
                  ELSE
                     SIGUIENTE<=ACTUAL;
                 END IF;



               WHEN IterandoZ=>

                 IF contador_i=32 THEN
                   SIGUIENTE<=Calculo_Z_final;
                  ELSE
                      SIGUIENTE<=PasoZaY0;
                 END IF;

               WHEN PasoZaY0=>

                   SIGUIENTE<=IterandoY;


               WHEN IterandoY=>

                  SIGUIENTE<=IterandoZ;


               WHEN Calculo_Z_final=>

                   SIGUIENTE<=Filtro_salida;


               WHEN Filtro_salida=>
                   fin<='1';
                   SIGUIENTE<=Reposo;
               END CASE;
            END PROCESS;
```

```
  --PROCESO SECUENCIAL PARA EL CONTADOR DE LAS 32 ITERACIONES

      PROCESS(reset,clk)
            BEGIN

            IF reset='1' THEN

                        contador_i <= 0;

              elsif clk'EVENT AND CLK='1' THEN
                IF ACTUAL=Reposo THEN
                    contador_i <= 0;

                    z <= x0;

            ELSIF ACTUAL=IterandoZ THEN

              IF  contador_i<32 THEN
                z <= shift_left(z,1) + shift_right((z + x"56AB0A"),1);
               ELSE
                contador_i <= 0;
              END IF;
            ELSIF ACTUAL=PasoZaY0 THEN
               y0 <= z;


            ELSIF ACTUAL=IterandoY THEN


                  y0 <= shift_right(y0,1) + shift_left(y0,1) + y0 + x1;
               contador_i <= contador_i + 1;


            ELSIF ACTUAL=Calculo_Z_final THEN
                        z <= z XOR y0;

               ELSIF ACTUAL=Filtro_salida THEN
                        z0 <= z(15 downto 0);  --me quedo con los 16 bits
menos significativos

                 END IF;
              END IF;
      END PROCESS;


  END opcionC;
```

Código empleado para N = 8 bits de salida (16 bits de entrada)

ejemplo del caso para el que surgen problemas en la simulación

```
library ieee;
    use ieee.std_logic_1164.all;
    use ieee.numeric_std.all;


--ENTIDAD

    ENTITY prngC IS
        GENERIC(N:INTEGER:=8);
        PORT(

            z0: OUT unsigned(N-1 DOWNTO 0);

           INICIO: IN STD_LOGIC;
            FIN: OUT STD_LOGIC;
            CLK,RESET: IN STD_LOGIC
            );
    END prngC;



--ARQUITECTURA

    architecture opcionC of prngC is


--DECLARACION DE SEÑALES
    TYPE ESTADO IS
(Reposo,IterandoZ,Auxiliar,PasoZaY0,IterandoY,Calculo_Z_final,Filtro_salida
);
    SIGNAL ACTUAL,SIGUIENTE:ESTADO;

    CONSTANT seed_0: unsigned((2*N-1) DOWNTO 0):= x"15B3"; --5.555 (n°s
decimales a usar
                                                           -- como semilla
para 16 bits
                                                           --de entrada)
    CONSTANT seed_1: unsigned((2*N-1) DOWNTO 0):= x"1E61"; --7.777

    CONSTANT x0: unsigned((2*N-1) DOWNTO 0):= x"F0E0";   --
x0=x0+((x0*x0)|5)
    CONSTANT x1: unsigned((2*N-1) DOWNTO 0):= x"FF2E";   --
x1=x1+((x1*x1)|13)

    SIGNAL z,y0: unsigned((2*N-1) DOWNTO 0);
    SIGNAL z_AUX: unsigned(23 DOWNTO 0); --Señal auxiliar para poder
emplear el
                                             --n° hexadecinal(0x56AB0A) al
ser de dimensiones
                                             --mayores a la señal a tratar
cuya dimensión es
                                             --el n° de bits a la entrada
    SIGNAL contador_i: INTEGER RANGE 0 TO 32;
    BEGIN
```

```
-- PROCESO PARA LA MAQUINA DE ESTADOS

 PROCESS(CLK,RESET)
             BEGIN

                  IF reset='1' THEN
                      actual<=reposo;

                   elsif clk 'EVENT AND clk='1' THEN
                       actual<=siguiente;

                   END IF;
               END PROCESS;


 --MAQUINA DE ESTADOS-----------------------------------------------


   PROCESS (ACTUAL,Inicio,contador_i)
       BEGIN

           CASE ACTUAL IS

                 WHEN Reposo=>

                   IF Inicio='1' THEN
                       SIGUIENTE<=IterandoZ;
                    ELSE
                       SIGUIENTE<=ACTUAL;
                   END IF;



                 WHEN IterandoZ=>

                   IF contador_i=32 THEN
                     SIGUIENTE<=Calculo_Z_final;
                    ELSE
                        SIGUIENTE<=Auxiliar;
                   END IF;

                 WHEN Auxiliar=>

                     SIGUIENTE<=PasoZaY0;

                 WHEN PasoZaY0=>

                     SIGUIENTE<=IterandoY;

                 WHEN IterandoY=>

                    SIGUIENTE<=IterandoZ;


                 WHEN Calculo_Z_final=>

                     SIGUIENTE<=Filtro_salida;
```

```
                        WHEN Filtro_salida=>
                            fin<='1';
                            SIGUIENTE<=Reposo;
                        END CASE;
                    END PROCESS;


  --PROCESO SECUENCIAL PARA EL CONTADOR DE LAS 32 ITERACIONES

      PROCESS(reset,clk)
            BEGIN

            IF reset='1' THEN

                        contador_i <= 0;

              elsif clk'EVENT AND CLK='1' THEN
                IF ACTUAL=Reposo THEN
                    contador_i <= 0;

                    z <= x0;

            ELSIF ACTUAL=IterandoZ THEN

              IF contador_i<32  THEN
                z_AUX <= shift_left(z,1) + shift_right((z + x"56AB0A"),1);
               ELSE
                contador_i <= 0;
              END IF;

            ELSIF ACTUAL=Auxiliar THEN
               z <= z_AUX(2*N-1 downto 0);

            ELSIF ACTUAL=PasoZaY0 THEN
               y0 <= z_AUX(2*N-1 downto 0);


            ELSIF ACTUAL=IterandoY THEN


                  y0 <= shift_right(y0,1) + shift_left(y0,1) + y0 + x1;
               contador_i <= contador_i + 1;


            ELSIF ACTUAL=Calculo_Z_final THEN
                        z <= z XOR y0;

               ELSIF ACTUAL=Filtro_salida THEN
                        z0 <= z(N-1 downto 0);  --me quedo con los 8 bits
menos significativos

                END IF;
             END IF;
      END PROCESS;


  END opcionC;
```

A.5 Códigos test-benchs empleados

Banco de pruebas para el algoritmo Blum Blum & Shub

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity bancoprueba is


end entity;

architecture a_bancoprueba of bancoprueba is


  component bbs
  GENERIC(N:INTEGER:=128);
       PORT(

            Semilla: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            M: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Salida: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
            Inicio: IN STD_LOGIC;
            Fin: OUT STD_LOGIC;
            CLK,RESET: IN STD_LOGIC;
            Fin_Termino_i: OUT STD_LOGIC
            );
  end component;



  signal reset, inicio, fin, fin_termino_i, clk: std_logic;
  signal M, Salida, Semilla: STD_LOGIC_VECTOR(127 DOWNTO 0);


begin

mi_test_BLUM: bbs port map
       (clk => clk,
        reset => reset,
        inicio => inicio,
        M => M,
        Semilla => Semilla,
        Salida => Salida,
        fin => fin,
        fin_termino_i => fin_termino_i);


-----------------------------------------
-- Ejemplo de banco de pruebas:
-----------------------------------------

--SIMULACION DEL RESET
```

```
      PROCESS
         BEGIN
            reset <= '0';
            INICIO<='0';
            WAIT FOR 150 ns;

                  reset <= '1';

            WAIT FOR 1000 ns;

                  reset <= '0';
                  INICIO<='1';

            WAIT FOR 10 us;

                  INICIO<='0';
            WAIT;
      END PROCESS;


-- SIMULACION CLK
      PROCESS
          BEGIN
              wait for 5 us;
              clk<='1';
              wait for 5 us;
              clk<='0';
          end process;

      PROCESS
          BEGIN

              Semilla<=CONV_STD_LOGIC_VECTOR(555,128);
              M<=CONV_STD_LOGIC_VECTOR(86909,128);
              WAIT;
          END PROCESS;

end a_bancoprueba;
```

Banco de pruebas para el algoritmo de la Opción A

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity testbench_PRNGs is

GENERIC(N:INTEGER:=16);

end entity;

architecture testbench of testbench_PRNGs is
```

```
  component prngA
  GENERIC(N:INTEGER:=16);
       PORT(

            z0: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);

            INICIO: IN STD_LOGIC;
            FIN: OUT STD_LOGIC;
            CLK,RESET: IN STD_LOGIC
            );
  end component;



  signal reset, inicio, fin, clk: std_logic;
  signal z0: STD_LOGIC_VECTOR(N-1 DOWNTO 0);


begin

mi_test_PRNG: prngA port map
       (clk => clk,
        reset => reset,
        inicio => inicio,
        z0 => z0,
        fin => fin);


-----------------------------------------
-- Ejemplo de banco de pruebas:
-----------------------------------------

--SIMULACION DEL RESET
      PROCESS
         BEGIN
            reset <= '0';
            INICIO<='0';
            WAIT FOR 150 ns;

                  reset <= '1';

            WAIT FOR 1000 ns;

                  reset <= '0';
                  INICIO<='1';

            WAIT FOR 10 us;

                  INICIO<='0';
            WAIT;
      END PROCESS;


-- SIMULACION CLK
      PROCESS
          BEGIN
              wait for 5 us;
              clk<='1';
              wait for 5 us;
```

```
            clk<='0';
        end process;


end testbench;
```

Banco de pruebas para los algoritmos de la Opción B y C

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity testbench_PRNGs is

GENERIC(N:INTEGER:=16);

end entity;

architecture testbench of testbench_PRNGs is


component prngB
GENERIC(N:INTEGER:=16);
PORT(

z0: OUT unsigned(N-1 DOWNTO 0);

INICIO: IN STD_LOGIC;
FIN: OUT STD_LOGIC;
CLK,RESET: IN STD_LOGIC
);
end component;


signal reset, inicio, fin, clk: std_logic;
signal z0: unsigned(N-1 DOWNTO 0);


begin

mi_test_PRNG: prngB port map
(clk => clk,
reset => reset,
inicio => inicio,
z0 => z0,
fin => fin);

-----------------------------------------
-- Ejemplo de banco de pruebas:
-----------------------------------------

--SIMULACION DEL RESET
PROCESS
```

```
BEGIN
reset <= '0';
INICIO<='0';
WAIT FOR 150 ns;

reset <= '1';

WAIT FOR 1000 ns;

reset <= '0';
INICIO<='1';

WAIT FOR 10 us;

INICIO<='0';
WAIT;
END PROCESS;


-- SIMULACION CLK
PROCESS
BEGIN
wait for 5 us;
clk<='1';
wait for 5 us;
clk<='0';
end process;



end testbench;
```

ANEXO B

Operadores a nivel de bit

Anexo B - Operadores a nivel de bit (bitwise operators)

Operador	Acción
&	AND a nivel de bit.
|	OR a nivel de bit.
^	XOR a nivel de bit.
~	Complemento.
<<	Desplazamiento a la izquierda.
>>	Desplazamiento a la derecha.

A continuación se describe cada uno de estos operadores brevemente:

DEFINICIÓN: **El operador AND (&)**: *Compara dos bits; si los dos son 1 el resultado es 1, en otro caso el resultado será 0. Ejemplo:*

```
c1 = 0x45       --> 01000101
c2 = 0x71       --> 01110001
--------------------------
c1 & c2 = 0x41 --> 01000001
```

DEFINICIÓN: **El operador OR (|)**: *Compara dos bits; si cualquiera de los dos bits es 1, entonces el resultado es 1; en otro caso será 0. Ejemplo:*

```
i1 = 0x47      --> 01000111
i2 = 0x53      --> 01010011
--------------------------
i1 | i2 = 0x57 --> 01010111
```

DEFINICIÓN: **El operador XOR (∧)**: *El operador OR exclusivo o XOR, dará como resultado un 1 si cualquiera de los dos operandos es 1, pero no los dos a la vez. Ejemplo:*

```
i1 = 0x47      --> 01000111
i2 = 0x53      --> 01010011
--------------------------
i1 ^ i2 = 0x14 --> 00010100
```

DEFINICIÓN: **El operador de complemento (~)**: *Este operador devuelve como resultado el complemento a uno del operando:*

```
c = 0x45  --> 01000101
----------------------
~c = 0xBA --> 10111010
```

DEFINICIÓN: **Los operadores de desplazamiento a nivel de bit (<< y >>)**: *Desplazan a la izquierda o a la derecha un número especificado de bits. En un desplazamiento a la izquierda los bits que sobran por el lado izquierdo se descartan y se rellenan los nuevos espacios con ceros. De manera análoga pasa con los desplazamientos a la derecha. Ejemplo:*

	c = 0x1C	00011100
c <<1	c = 0x38	00111000
c >>2	c = 0x07	00000111

ANEXO C

Comandos básicos del editor vi de los S.O. Unix

Anexo C - Comandos básicos del editor vi de la família de sistemas operativos Unix

Vi es el editor estándar de los Sistemas Operativos de la familia Unix/Linux. En principio puede resultar un poco difícil, sobre todo para los usuarios de Windows. Sin embargo, dado que es prácticamente el único editor que puedes encontrar pre-instalado en cualquier PC o Servidor con Unix/Linux, aprender a usarlo es absolutamente indispensable.

.==============================.
1.=|«««««««« INVOCACIÓN VI »»»»»»»»|
'==============================='

$vi----------Editar un texto sin nombre
$vi archivo----------Editar un archivo (nuevo o no)
$vi archivo1 archivo2------Editar lista de archivos
$vi +n archivo----------Editar archivo en la línea n
$vi +/txt archivo----------Editar archivo en la 1a línea donde aparece txt

.======================================.
2.=|«««««««« MOVIMIENTOS DEL CURSOR »»»»»»»»|
'======================================='

Arriba----k
Abajo-----j
Derecha---h
Izquierda-l

0---------Inicio de línea
$---------Fin de línea
w---------Word: Avanzar palabra
b---------Back: Retroceder palabra
e---------End: Al final de palabra
H---------Home: Esquina sup. izq. de la ventana
L---------Last: Esquina inf. izq. de la ventana
crtl+u----Window up: Subir ventana
crtl+d----Window down: Bajar ventana
crlt+b----Page back: Retroceder página
crtl+f----Page forward: Avanzar página
nG--------Go: Salta a la línea n.
1G--------A la primera línea
$G--------A la última línea
fcar------Buscar en la línea el carácter car (hacia delante)
Fcar------Buscar en la línea el carácter car (hacia atrás)

.================================.
3.=|«««««««« INSERTAR TEXTO »»»»»»»»|
'================================'
i---Insertar (delante del cursor)
I---Insertar al principio de la línea
a---Añadir (detrás del cursor)
A---Añadir al final de la línea
o---Insertar una línea debajo de la actual
O---Insertar una línea encima de la actual

.================================.
4.=|«««««««« BORRAR TEXTO »»»»»»»»|
'================================'

x---Borrar caracter actual
X---Borrar caracter anterior
dd--Borrar línea actual
D---Borrar hasta final de línea
dw--Borrar palabra

.================================.
5.=|«««««««« CAMBIAR TEXTO »»»»»»»»|
'================================'

rcar--Reemplazar el caracter actual por car
R-----Reemplazar texto desde la posición del cursor
s-----Substituir el caracter actual por texto a insertar
S-----Substituir la línea actual
C-----Cambiar hasta el final de la línea
cw----Cambiar palabra
J-----Unir a la línea actual la siguiente

.================================.
6.=|«««««««« COPIAR Y PEGAR »»»»»»»»|
'================================'

yy----Copiar en el buffer la línea actual
nyy---Copiar en el buffer n líneas desde la actual
p-----Pega el buffer detrás del cursor
P-----Pega el buffer delante del cursor

.====================================.
7.=|«««««««« BUSCAR Y SUBSTITUIR »»»»»»»»|
'====================================='

%------------------Busca el caracter delimitador ( ) [ ] { } que balancea el actual (Dentro de un entorno salta al

delimitador inicial)
/ExpReg-------------Busca hacia delante la expresión regular ExpReg
?ExpReg-------------Busca hacia atrás la expresión regular ExpReg
n-------------------Repite la última búsqueda

N----------Repite la última búsqueda en el sentido contrario
:s/txt/txt2---------Substituye el texto txt por txt2 la primera vez que aparece en la línea
:s/txt/txt2 /g------Substituye todas las apariciones de txt por txt2 en la línea
:m,n s/txt/txt2 /g--Substituye en el rango de líneas [m,n]

.====================================.
8.=|«««««««« REPETIR Y DESHACER »»»»»»»»|
'====================================='

.---Repetir último comando de actualización (Borrado/Inserción/Cambio)
u---Deshacer último comando de actualización
U---Deshacer todos los cambios en la línea actual

.====================================.
9.=|«««««««« COMANDOS DEL SHELL »»»»»»»»|
'====================================='

:sh----------Invoca un nuevo shell. Al salir continua la edición
:!CmdShell----Ejecuta un comando del sistema operativo
:r!CmdShell---Ejecuta un comando del S.O. e inserta su salida en la posición del cursor
:!!----------Repite el último comando ejecutado en un shell

.==.
10.=|«««««««« OPERACIONES CON ARCHIVOS »»»»»»»»|
'=='

:w----Graba las modificaciones efectuadas en el archivo
:w----archivo Escribe el texto actual en archivo (Sólo si no existía)
:q----Salir (si no hay cambios)
:q!---Salir (sin grabar)
:wq---Guardar cambios y salir
:x----Guardar cambios y salir
ZZ----Guardar cambios y salir

.==.
11.=|«««««««« ESTADISTICAS DE ARCHIVO »»»»»»»»|
'=='

:=------Muestra el número total de líneas del archivo
:.=-----Muestra el número de línea actual
crtl+G--Muestra el nombre del archivo, línea actual, número total de líneas y porcentaje recorrido del archivo.

.====================================.
12.=|«««««««« OPCIONES DE ENTORNO »»»»»»»»|
'====================================='

:set opción-----Activa la opción de vi correspondiente
:set noopción---Desactiva la opción de vi correspondiente

all-------Muestra todas las opciones y sus valores
number----Muestra numeración de líneas

list------Muestra caracteres de control
ic--------Ignora mayúsculas/minúsculas en las búsquedas